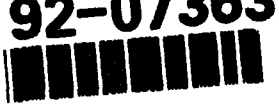
(2)

Annual Report for Contract Number N00014-88-K-0641

For the period: 1 October 1990 - 30 September 1991



1-57234

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 90 - 30 Sep 91

1 Productivity Measures

Refereed papers submitted but not yet published: 0
Refereed papers published: 0
Unrefereed reports and articles: 2
Books or parts thereof submitted but not yet published: 0
Books or parts thereof published: 1
Patents filed but not yet granted: 0
Patents granted: 0
Invited presentations: 0
Honors received (fellowships, technical society appointments, conference committee role, editorship, etc): 0
Prizes or awards received: 0
Promotions obtained: 0
Graduate students supported >= 25% of full time: 1
Post-docs supported >= 25% of full time: 0
Minorities supported: 0

Statement A per telecon
Dr. Andre Van Tilborg ONR/Code 1133
Arlington, VA 22217-5000

NWW 4/10/92

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 90 - 30 Sep 91

2 Detailed Summary of Technical Progess

This section presents background information, motivation, and a project description of my current focus of research, ***type evolution in the presence of persistent instances.***

Background

Over the past decade there have been a number of research projects dealing with the integration of database features into programming language systems. This was done in an effort to simplify the programming of applications requiring such features. For example, efforts aimed at developing programming languages for reliable, fault-tolerant applications have imported transactions and data recovery techniques from the database world [14, 18, 6]. At the same time, other projects have worked to incorporate a notion of persistent data into programming languages, in order to support applications requiring one or more of the following features:

- Manipulation of very large amounts of data
- Sharing of data among application programs
- Sharing of data among users
- Persistence of data across program invocations

Design applications, exemplified by CAD/CAM systems, multimedia and office automation facilities, and software engineering systems, typically require all of these features, but unlike traditional database applications, require a traditional computational facility as well. For this reason, available database facilities, such as relational database management systems (RDBMS) are not acceptable. To this end, a number of research projects have developed programming

language systems that incorporate a notion of data persistence. A detailed survey of persistent programming languages can be found in [5]. A survey of the issues related to persistence and typing can be found in [1].

The first interfaces between programming languages and database systems were made up of subroutine libraries, enabling programs to pass commands, in the form of strings, to the process managing the database. Values of certain types could be communicated between the database and the program via a predefined set of variables. For applications requiring considerable database interaction at runtime, this communications medium was clumsy and inconvenient. Also, the forms of data that could be entered into the database was typically very primitive, restricted to tuples of strings and numbers. Early *persistent programming languages*[1] concentrated on bridging the gap in form between data in the program and in the database.

The first such persistent programming languages were designed as extensions to existing Algol-like languages [17, 2]. In them, persistent data was stored in disk-based objects that were manipulated very much like traditional files. In PASCAL-R [17], these data repositories were typed similarly to PASCAL files, restricted to contain a homogeneous collection of *relation*[2] records. A number of control structures were provided for simple iteration and querying over the elements of the database file. Applications in PASCAL-R, forced to conform to the restrictive range of persistent data types, had to devote time converting between the runtime and persistent representations of the data. This characteristic, called *impedance mismatch* by a number of researchers, increased application development time, and made the language inelegant.

PS-Algol [2], on the other hand, provided a model of persistent data that was more integrated with the base language. All objects in a database file were reachable by way of pointer reference from an easily accessible root. In addition, any PS-Algol object could be stored in the database (not just simple relations as in PASCAL-R). This improvement, called *orthogonal persistence*, made PS-Algol much more interesting and useful than its predecessors.

With orthogonal persistence a desirable feature, many projects have found object-oriented languages an appropriate platform for research, their data models and type systems being more closely related to the database and semantic models than those of the Algol-like languages. Object-oriented systems are

[1] A *persistent programming language* is a language that provides to its clientele the ability to preserve data across successive executions of a program, and even allows such data to be used by many different programs. Data in a persistent programming language is independent of any program, able to exist beyond the execution and lifetime of the code that created it.

A *database programming language* is a language that integrates some ideas from the database programming model with traditional programming language features. Such a language is distinguished from a persistent programming language in that it incorporates features beyond persistence, such as transactions, locking, and query processing.

[2] The same type of object as is found in relational database systems.

characterized by the following features: object identity, abstract data typing, inheritance (or subtyping), late (*i.e.*, runtime) binding of methods to objects. It turns out that these features are quite attractive to the developer of a design database system, as they map quite well to the mental framework of the design engineer [11]. The object-oriented data model is compatible with the **Semantic Data Model** [16], which requires support for instantiation, aggregation, and generalization, all of which are offered by the object-oriented system's notion of object classes and inheritance. It has also been shown that some other database features, such as composite objects, object versioning, *etc.* can be built out of the intrinsic features of an object-oriented database system [13].

Object-oriented database systems have a computation model better suited for computationally extensive tasks requiring persistent data than the older database systems (such as relational databases) do. The older systems are optimized for queries over large collections of data, and provide little support for much else. The object-oriented model allows for efficient navigational access (*i.e.*, browsing) over individual objects, and by associating procedural information with types, allows for a wider range of functionality.

Examples of recently-developed object-oriented database systems include ORION [13], GemStone [15, 3], Encore [12], and O_2 [7].

The Problem

Programming is an incremental process. Frequently, existing program elements are enhanced to support additional functionality or a new application. Such an operation often necessitates the propagation of changes to dependent elements throughout the program. ***Abstract data typing*** was developed as a way of building logical barriers between the specification of an interface to a type, and the implementation of that interface and its internal representation. This modularity ***fire wall*** effectively contains (type) implementation changes from the various program elements which depend on the type. However, modularity does not address the problem of modification to type specifications. In general, the effects of an ***evolution*** of a type cannot be locally contained [8].

Type evolution is the term used to describe the process of altering type definitions over time. It encompasses the problem of specifying the change, as well as attempts to manage the affects of the evolution on dependent elements. In an object-oriented system, types are implemented as ***classes***, defined by a representation and a set of (procedural) access methods. Type (or class) evolution in such a context is concerned with the propagation of class changes to associated methods, and to the subclasses inheriting part of their definitions from the parent class.

By way of example, consider the database of phone numbers managed the local telephone company. The database associates phone number with customer, billing address, service level, touch-tone vs. rotary dial, etc. Now, it is 1983,

and the FCC requires the local phone company to associate an additional data field, a default long-distance carrier, with each number in the database. Since it obviously cannot afford to discard its records, the phone company must develop a method for adjusting its database to cope with the creation and definition of the additional field, as well as modify its database applications to recognize and accept the new schema. Such is an example of type evolution in the presence of persistent data.

While type evolution in traditional programs need only be concerned with the effect of the change on code, the persistent data associated with a database program, is another dependency that much be addressed.

It is also worth mentioning that the type modularity barrier resulting from abstraction fails in the context of persistent data, which unlike non-local program references, depends on the representation of the type. Barring some special system support for change (as discussed below), those instances would have to be modified or discarded, so as to conform to the (new) class specification.

Projects and Goals

In the realm of database programming languages, it is very hard to escape the repercussions of change. Even the most traumatic type change cannot effect previously-compiled programs when those programs are run in isolation. Database applications, by definition, manipulate shared data. A change effecting the definition of a type that is represented in the database renders old database program images obsolete. One major goal of this, my dissertation project [4], is to examine the issues related to the goal of keeping the (shared) data consistent and accessible in the presence of type evolution.

Existing database programming systems can be classified into three categories, according to how they handle old instances in the presence of a change to the database schema:

1. **No support.** Either the modified type is distinct from the original and old instances are not accessible as new instances, or the new type is the same as the new, and serious errors result if old database items are referenced.

2. **Database conversion.** Instances of the original type are converted into instances of the new type. Often evolutions are restricted to those for which the system can determine how to convert the types, but sometimes the programmer can supply conversion routines. The database instances themselves can be converted at evolution time, or, as is more often the case, when they are next accessed.

3. **Emulation.** Instances of the original type persist indefinitely, the system instead insulating the applications expecting revised instances by providing a layer between the old objects and the new application. This layer emulates the revised object semantics on top of the old object format.

This approach has the significant advantage of not rendering old applications obsolete, as they can continue to access the data through the old type interface.

One of the basic premises of my perspective is that neither of the two active support strategies (*i.e.*, conversion or emulation) is adequate for all circumstances. In situations where there are few applications (and copies of application images), conversion is the preferable approach, as it is more efficient. However, a site might decide that it wants to bear the price of emulation in cases where it is impractical or impossible to upgrade all the applications, or when multiple interfaces to the data is desirable (*e.g.*, in a design system where multiple users are typically modifying the database schema). One approach might be to provide both mechanisms, and allow the programmer to select which strategy is the more appropriate. While such a scheme is an improvement, it is still insufficient, as there exist other strategies that would be more appropriate in certain circumstances.

Imagine a type change that consists of a modification to a single attribute of a type. Imagine that the conversion process would take a considerable amount of time per object.

If our system supports conversion, the first access to any particular object after the installation of the change will be delayed until the database server can convert the object. This penalty would be paid even if the client is not making use of this modified attribute.

If our system supports emulation, then the client will only have to the (time) cost of conversion when he needs the value of the modified attribute. However, unlike conversion, that cost will be paid on *every* access, not just the first.

If we had a system which supported *both* conversion *and* emulation, then the server could "emulate" the object on accesses until the first reference to the new attribute is made. At that time, the object could be converted. This strategy is preferable than either conversion of emulation alone, because it avoids an expensive conversion when the user does not require the results.

One further enchancement in a system supporting both emulation and conversion could be to emulate during the day, and convert objects incrementally at night, when there are cycles to spare.

A preferable general solution would be to combine the compatibility support of emulation with the efficiency of conversion. Whether this is possible in general is one of the project's topics of research. Under certain conditions, hybrid strategies more sophisticated than the one presented above might make better use of the database servers time than any approach presented to date.

Tuning performance by playing with the time before which an object can be converted can only take you so far. In general, the granularity of the options (either the object is an instance of the old type-version or it is an instance of the new) is too large. Better performance is likely possible for some mix of applications if it were possible to somehow *merge* the specifications of the two (or more) type-versions, supporting both backward and forward compatibility at a (minimized) cost to both old and new clients.

In fact, we do not have to wish for this possibility for too long as there exists a technology that addresses this very issue. That technology is *view-based abstraction.*

Views[9, 10] are a mechanism for supporting multiple external specifications on an abstract data type. The resulting system supports instances that may be operated upon by different applications using different type specifications, but which are all operating on a common instance, whose concrete implementation represents all the **facets** (*i.e.*, views of an instance) of the object.

A significant class of hybrid evolution support strategies can thus be represented as a view implementation problem, with full compatibility supported though various views of a type (one for each active type-version).

Views were initially devised for situations where all the various interfaces were known, prior to the creation (or persistence) of any instances. For views to be useful as a type evolution mechanism, they will have be extended to deal with 1) the dynamic addition of interfaces, and 2) the existence of objects represented in multiple ways. In the context of an object-oriented data model, views will also have to be adapted to cope with class inheritance. How this will be accomplished is a project research goal. Initial approaches to this task are included in the author's dissertation proposal [4].

This concludes the presentation of the perspective and goals of the ongoing project. The following section lists related work done in the past year.

Past Accomplishments

My research into this field begun by conducting a detailed survey of the field of persistent programming languages, programming languages which include a notion of data persistence. The resulting survey can be found in [5].

Of direct relevance to my dissertation project, progress has already been made in the following areas:

- Development of a cost model to evaluate various evolution support strategies
- Research into real world expense of evolution management

If our system is to support multiple compatibility and conversion strategies, some method of performance evaluation is necessary to assist in the selection of

the appropriate strategy. To this end, a simple cost model has been developed. Currently, the model has only been used to compare the various basic strategies (*i.e.*, those supported by existing systems), but it is intended that the model be generalized.

In order to gather information concerning the costs and importance of evolution management under real world conditions, it is hoped that actual cases can be studied. A few inquiries have been made to date, and many more are expected, in pursuit of this goal.

Future Goals

In order to achieve our final goal of a flexible, customizable evolution management system for database programming, the following tasks need to be performed:

- Extend model of views to cope with multiple representations, inheritance, and concurrent evolutions.
- Develop prototype system on top of a real database system.
- Develop/implement library of hybrid strategies.
- Enhance and then use cost model to think about applicability of various strategies in various database applications.
- Experiment with type evolution support in a particular application domain

Current views technology only addresses the issue of multiple, static interfaces to a type. To be useful in our situation, the views model must be extended to cope with:

1. Dynamic creation of views (corresponding to type evolutions after the instantiation of (older) versions of the type).
2. Multiple representations of instances in the database
3. Inheritance (in the case of object-oriented databases).
4. Concurrent evolutions

In order to test and extend our initial ideas, a prototype database system with advanced evolution management must be designed and built. Rather than create a database system from scratch, the management subsystem will be built on top of an existing database. It is expected, however, that the data model exported by the underlying database system will have to be masked from the user.

In this way, a consistent data model with evolution support can be presented to the user.

The combined emulation-and-conversion strategy presented earlier (p. 6) is but one example of a hybrid strategy. We expect many others to be developed and evaluated (using both the cost model and experimental measurement) in the course of our research.

In order to attract people to make use of the facility, our evolution management system will be integrated with the various existing database systems of a number of projects on campus. In this way, researchers requiring the use of a database for their applications can benefit from our work, as well as provide important feedback on the utility of our system.

Concluding Remarks

The goal of my dissertation research project is to explore the dynamics of schema evolution management in situations where there exists an associated database of considerable size and value. Changes to the database schema are usually motivated by a desire to increase the functionality of one or more of the applications operating on the database. However, such modifications (to the schema and to the application) cannot be done in isolation, as they might require corresponding changes to be made to both the other database applications and to the database itself.

In the course of research, a prototype system will be constructed exhibiting some of the features described earlier in this report. It is intended that this system will be used by a number of independent projects requiring database support, in order to provide us with important feedback.

References

[1] Atkinson, M. P. and Buneman, O. P. *Types and Persistence in Database Programming Languages.* **ACM Computing Surveys**, vol. 19 (1987), pp. 105–190.

[2] Atkinson, M., Bailey, P., Chisolm, K., Cockshott, W., and Morrison, R. *An Approach to Persistent Programming.* **Computer Journal**, vol. 26 (1983), pp. 360–365.

[3] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. *The GemStone Data Management System.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[4] Clamen, S. *Type Evolution in the Presence of Persistent Instances.* August 1991. *Thesis Proposal.*

[5] Clamen, S. M. *Data Persistence in Programming Languages - A Survey.* Technical Report, no. CMU-CS-91-155, Cambridge, MA, Pittsburgh, PA, May 1991.

[6] Detlefs, D., Herlihy, M. P., and Wing, J. M. *Inheritance of Synchronization/Recovery Properties in Avalon/C++.* **Computer**, vol. 21 (1988).

[7] Deux, O. and *et. al. The Story of O_2.* **IEEE Transactions on Knowledge and Data Engineering**, vol. 2 (1990), pp. 91–108.

[8] Garlan, D., Kaiser, G., and Notkin, D. *On the Criteria To Be Used in Composing Tools into Systems.* Technical Report, no. TR 88-08-09, University of Washington, Seattle, WA 98185, August 1988.

[9] Garlan, D. B. *Views for Tools in Integrated Environments*, Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 1987. *Available as Technical Report CMU-CS-87-147.*

[10] Habermann, A. N. *et. al.. Programming with Views.* no. CMU-CS-87-177, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, 1988.

[11] Heiler, S., Dayal, U., Orenstein, J., and Radke-Sproull, S. *An Object-Oriented Approach to Data Management: Why Design Databases Need It.* in: **Proceedings of the 14th ACM/IEEE Design Automation Conference**. 1987, pp. 335–340.

[12] Hornick, M. F. and Zdonik, S. B. *A Shared, Segmented Memory System for an Object-Oriented Database.* **ACM Transactions on Office Information Systems**, vol. 5 (1987), pp. 70–95.

[13] Kim, W., Ballou, N., Chou, H.-T., Garza, J., Woelk, D., and Banerjee, J. *Integrating an Object-Oriented Programming System with a Database System.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)**. San Diego, CA, 1988, pp. 142–152.

[14] Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W. *Argus Reference Manual.* no. TR-400, MIT Laboratory for Computer Science, Cambridge. MA, November 1987.

[15] Maier, D. and Stein, J. *Development and Implementation of an Object-Oriented DBMS.* **MIT Press Series in Computer Systems**, MIT Press, Cambridge, MA, 1987, pp. 355–392.

[16] McCleod, D. and King, R. *Semantics Database Models.* in: **Principles of Database Design**, edited by S. Yao. Prentice-Hall, 1984.

[17] Schmidt, J. W. *Some High Level Language Constructs for Data of Type Relation.* **ACM Transactions on Database Systems**, vol. 2 (1977), pp. 247–261.

[18] Spector, A. Z., Bloch, J. J., Daniels, D. S., Draves, R. P., Duchamp, D., Eppinger, J. L., Menees, S. G., and Thompson, D. S. *The Camelot Project.* **Database Engineering**, vol. 9 (1986). *Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.*

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
nh@cs.cmu.edu
ONR Funding
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 90 - 30 Sep 91

3 Publications, Presentations and Reports

Below are the list of publications I have been involved with during the past year:

[1] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietske, Richard Lerner, and Su Yuen Ling. The Avalon language. In Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*, The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

[2] Stewart M. Clamen. Data persistence in programming languages – a survey. Technical Report CMU-CS-91-155, Cambridge, MA, Pittsburgh, PA, May 1991.

[3] Stewart Clamen. Type evolution in the presence of persistent instances. Thesis Proposal, August 1991.

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 90 - 30 Sep 91

4 Research Transitions and DoD Interactions

As part of my plan of research, I have made arrangements with a number of independent research organizations here at Carnegie Mellon to make direct use of my prototype system in their research projects. I have made such arrangements for two reasons:

- To provide me with feedback about the utility and applicability of my work.
- To provide a useful facility for database management in the respective applications.

References

[1] Atkinson, M. P. and Buneman, O. P. *Types and Persistence in Database Programming Languages.* **ACM Computing Surveys**, vol. 19 (1987), pp. 105–190.

[2] Atkinson, M., Bailey, P., Chisolm, K., Cockshott, W., and Morrison, R. *An Approach to Persistent Programming.* **Computer Journal**, vol. 26 (1983), pp. 360–365.

[3] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. *The GemStone Data Management System.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[4] Clamen, S. *Type Evolution in the Presence of Persistent Instances.* August 1991. *Thesis Proposal.*

[5] Clamen, S. M. *Data Persistence in Programming Languages – A Survey.* Technical Report, no. CMU-CS-91-155, Cambridge, MA, Pittsburgh, PA, May 1991.

[6] Detlefs, D., Herlihy, M. P., and Wing, J. M. *Inheritance of Synchronization/Recovery Properties in Avalon/C++.* **Computer**, vol. 21 (1988).

[7] Deux, O. and *et. al. The Story of O_2.* **IEEE Transactions on Knowledge and Data Engineering**, vol. 2 (1990), pp. 91–108.

[8] Garlan, D., Kaiser, G., and Notkin, D. *On the Criteria To Be Used in Composing Tools into Systems.* Technical Report, no. TR 88-08-09, University of Washington, Seattle, WA 98185, August 1988.

[9] Garlan, D. B. *Views for Tools in Integrated Environments*, Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 1987. *Available as Technical Report CMU-CS-87-147.*

[10] Habermann, A. N. *et. al.. Programming with Views.* no. CMU-CS-87-177, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, 1988.

[11] Heiler, S., Dayal, U., Orenstein, J., and Radke-Sproull, S. *An Object-Oriented Approach to Data Management: Why Design Databases Need It.* in: **Proceedings of the 14th ACM/IEEE Design Automation Conference.** 1987, pp. 335–340.

[12] Hornick, M. F. and Zdonik, S. B. *A Shared, Segmented Memory System for an Object-Oriented Database.* **ACM Transactions on Office Information Systems**, vol. 5 (1987), pp. 70–95.

[13] Kim, W., Ballou, N., Chou, H.-T., Garza, J., Woelk, D., and Banerjee, J. *Integrating an Object-Oriented Programming System with a Database System.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)**. San Diego, CA, 1988, pp. 142–152.

[14] Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W. *Argus Reference Manual.* no. TR-400, MIT Laboratory for Computer Science, Cambridge, MA, November 1987.

[15] Maier, D. and Stein, J. *Development and Implementation of an Object-Oriented DBMS.* **MIT Press Series in Computer Systems**, MIT Press, Cambridge, MA, 1987, pp. 355–392.

[16] McCleod, D. and King, R. *Semantics Database Models.* in: **Principles of Database Design**, edited by S. Yao. Prentice-Hall, 1984.

[17] Schmidt, J. W. *Some High Level Language Constructs for Data of Type Relation.* **ACM Transactions on Database Systems**, vol. 2 (1977), pp. 247–261.

[18] Spector, A. Z., Bloch, J. J., Daniels, D. S., Draves, R. P., Duchamp, D., Eppinger, J. L., Menees, S. G., and Thompson, D. S. *The Camelot Project.* **Database Engineering**, vol. 9 (1986). *Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.*

Managing Type Evolution in the Presence of Persistent Instances (Thesis Proposal)

Stewart M. Clamen
School of Computer Science
Carnegie Mellon University

August 15, 1991

Abstract

A longstanding problem in the realm of database applications is how to manage the data dependencies associated with a change in the database schema definition. More specifically, how can a programmer modify the definition or implementation of a data type without abandoning the instances of the current type which persist in the database? While adequate solutions exist for relational database systems, the problem is exacerbated in the new *database programming languages*, where more elaborate type specifications are supported.

This proposal begins by exploring the various solutions to the problem, examining how existing database programming systems cope with *type evolution* and database consistency. It then motivates and sketches a general and extensible framework for the management of type evolution and its effects. Such a framework would provide the programmer complete decision-making power over the conversion of existing instances, thereby giving control over the obsolescence of application programs which depend on outdated data type specifications.

1 Background: Integrating Database Technology and Programming Languages

Over the past decade there have been a number of research projects dealing with the integration of database features into programming language systems. This was done in an effort to simplify the programming of applications requiring such features. For example, efforts aimed at developing programming languages for reliable, fault-tolerant applications have imported transactions and data recovery techniques from the database world [32, 43, 15]. At the same time, other projects have worked to incorporate a notion of persistent data into programming languages, in order to support applications requiring one or more of the following features:

- Manipulation of very large amounts of data
- Sharing of data among application programs
- Sharing of data among users
- Persistence of data across program invocations

Design applications, exemplified by CAD/CAM systems, multimedia and office automation facilities, and software engineering systems, typically require all of these features, but unlike traditional database applications, require a traditional computational facility as well. For this reason, available database facilities, such as relational database management systems (RDBMS) are not acceptable. To this end, a number of research projects have developed programming language systems that incorporate a notion of data persistence.[1] The remainder of this section presents a brief history of the development of such language systems.

The first interfaces between programming languages and database systems were made up of subroutine libraries, enabling programs to pass commands, in the form of strings, to the process managing the database. Values of certain types could be communicated between the database and the program via a predefined set of variables. For applications requiring considerable database interaction at runtime, this communications medium was clumsy and inconvenient. Also, the forms of data that could be entered into the database was typically very primitive, restricted to tuples of strings and numbers. Early *persistent programming languages*[2] concentrated on bridging the gap in form between data in the program and in the database.

The first such persistent programming languages were designed as extensions to existing Algol-like languages [40, 3]. In them, persistent data was stored in disk-based

[1] A detailed survey of persistent programming languages can be found in [12]. A survey of the issues related to persistence and typing can be found in [2].

[2] A *persistent programming language* is a language that provides to its clientele the ability to preserve data across successive executions of a program, and even allows such data to be used by many different programs. Data in a persistent programming language is independent of any program, able to exist

objects that were manipulated very much like traditional files. In PASCAL-R [40], these data repositories were typed similarly to PASCAL files, restricted to contain a homogeneous collection of *relation*[3] records. A number of control structures were provided for simple iteration and querying over the elements of the database file. Applications in PASCAL-R, forced to conform to the restrictive range of persistent data types, had to devote time converting between the runtime and persistent representations of the data. This characteristic, called *impedance mismatch* by a number of researchers, increased application development time, and made the language inelegant.

PS-Algol [3], on the other hand, provided a model of persistent data that was more integrated with the base language. All objects in a database file were reachable by way of pointer reference from an easily accessible root. In addition, any PS-Algol object could be stored in the database (not just simple relations as in PASCAL-R). This improvement, called *orthogonal persistence*, made PS-Algol much more interesting and useful than its predecessors.

With orthogonal persistence a desireable feature, many projects have found object-oriented languages an appropriate platform for research, their data models and type systems being more closely related to the database and semantic models than those of the Algol-like languages. Object-oriented systems are characterized by the following features: object identity, abstract data typing, inheritance (or subtyping), late (*i.e.*, runtime) binding of methods to objects. It turns out that these features are quite attractive to the developer of a design database system, as they map quite well to the mental framework of the design engineer [24]. The object-oriented data model is compatible with the **Semantic Data Model** [34], which requires support for instantiation, aggregation, and generalization, all of which are offered by the object-oriented system's notion of object classes and inheritance. It has also been shown that some other database features, such as composite objects, object versioning, *etc.* can be built out of the instrinsic features of an object-oriented database system [29].

Object-oriented database systems have a computation model better suited for computationally extensive tasks requiring persistent data than the older database systems (such as relational databases) do. The older systems are optimized for queries over large collections of data, and provide little support for much else. The object-oriented model allows for efficient navigational access (*i.e.*, browsing) over individual objects, and by associating procedural information with types, allows for a wider range of functionality.

Examples of recently-developed object-oriented database systems include ORION [29], GemStone [33, 10], Encore [26], and O_2 [16].

---

beyond the execution and lifetime of the code that created it.

A *database programming language* is a language that integrates some ideas from the database programming model with traditional programming language features. Such a language is distinguished from a persistent programming language in that it incorporates features beyond persistence, such as transactions, locking, and query processing.

[3]The same type of object as is found in relational database systems.

2 The Problem: Type Evolution in the Presence of Persistent Data

This section describes the problem that the thesis will research. It examines existing solutions to the problem and demonstrates their shortcomings.

2.1 What is Type Evolution?

Programming is an incremental process. Frequently, existing program elements are enhanced to support additional functionality or a new application. Such an operation often necessitates the propagation of changes to dependent elements throughout the program. *Abstract data typing* was developed as a way of building logical barriers between the specification of an interface to a type, and the implementation of that interface and its internal representation. This modularity *fire wall* effectively contains (type) implementation changes from the various program elements which depend on the type. However, modularity does not address the problem of modification to type specifications. In general, the effects of an *evolution* of a type cannot be locally contained [20].

Type evolution[4] is the term used to describe the process of altering type definitions over time. It encompasses the problem of specifying the change, as well as attempts to manage the affects of the evolution on dependent elements. In an object-oriented system, types are implemented as *classes*, defined by a representation and a set of (procedural) access methods. Class evolution in such a context is concerned with the propogation of class changes to associated methods, and to the subclasses inheriting part of their definitions from the parent class. A list of common evolutions is included in Figure 1 (p.4).

By way of example, consider the class hierarchy for a hypermedia system presented in Figure 2 (p.5). The evolution, as depicted in Figure 3, involves the change of the domain of the public `DisplayFont` attribute from `string` to `font`, affects all code elements which refer to the domain of the `DisplayFont` attribute. This includes to the (local) class methods of `TextWindow` (such as the `Get_DisplayFont` and `Set_DisplayFont` methods that implement the public attribute) as well as non-local methods in inherited or dependent classes.

The problem as described so far is an issue in all program development systems. Type abstraction and the modularity it affords protects against implementation changes, but does not address the problem when the specification of the type interface changes.

[4] The synonymous terms *schema evolution* and *class evolution* are also often used, depending on the language context. **Class evolution** is used in the context of object-oriented language models, where types are implemented as *classes*. **Schema evolution** is an older, database systems term, where schema refers to the format of the data. *Type evolution* is the term used most frequently by the author, although it should be noted that he does exhibit a tendency to use *schema evolution* in contexts where he wants to attract the attention of database people.

(1) Changes to the contents of a node (a class)

  (1.1) Changes to an instance variable

    (1.1.1) Add a new instance variable to a class

    (1.1.2) Drop an existing instance variable from a class

    (1.1.3) Change the name of an instance variable of a class

    (1.1.4) Change the domain of an instance variable of a class

    (1.1.5) Change the inheritance (parent) of an instance variable (*i.e.*, inherit another instance variable with the same name).

    (1.1.6) Change the default value of an instance variable

    (1.1.7) Manipulate the shared value of an instance variable

      (1.1.7.1) Add a shared value

      (1.1.7.2) Change the shared value

      (1.1.7.3) Drop the shared value

    (1.1.8) Drop the composite link property on an instance variable.[a]

  (1.2) Changes to a method

    (1.2.1) Add a new method to a class

    (1.2.2) Drop an existing method from a class

    (1.2.3) Change the name of a method of a class

    (1.2.4) Change the code of a method of a class

    (1.2.5) Change the inheritance (parent) of an method (*i.e.*, inherit another method with the same name).

(2) Changes to an edge (on the class-inheritance hierarchy)

  (2.1) Make a class S a superclass of a class C

  (2.2) Remove a class S from the superclass list of a class C

  (2.3) Change the order of superclasses of a class C

(3) Changes to a node (on the class-inheritance hierarchy)

  (3.1) Add a new class

  (3.2) Drop an existing class

  (3.3) Change the name of a class

[a] This relates to ORION's composite object facility.

Figure 1: List of evolutions on classes and class hierarchy supported by the ORION object-oriented database system. Some of these evolutions refer to data model features particular to ORION (such as the composite objects, multiple inheritance) which not be supported in other object-oriented systems.

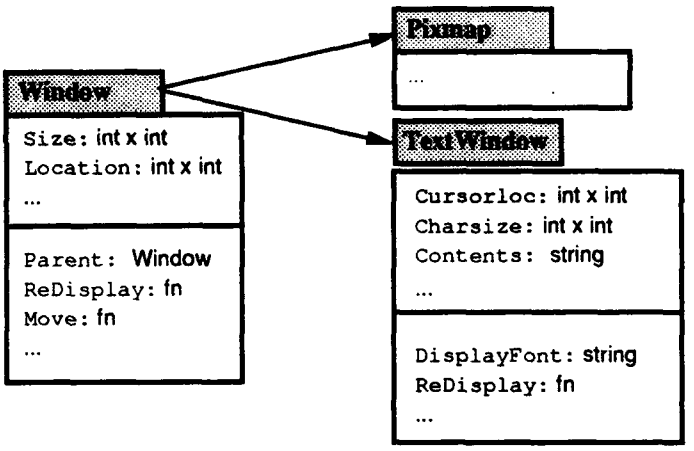


Figure 2: A sample hierarchy of classes, to be used to illustrate some examples of type evolution. The line within the class definitions distinguishes private from public features.

For this reason, type evolution support is an issue in all development systems, and is a topic of research in many software engineering projects [44, 20, 11, 22].

In database programming systems, the repercussions of a type evolution are more severe. Type evolution in traditional programs need only be concerned with affect of the change on code. The persistent data associated with a database program, however, is another dependency that much be addressed. In the context of our example, the change to the type of `DisplayFont` would render all persistent instances of `TextWindow` inconsistent with the new definition.

It is also worth mentioning that the type modularity barrier resulting from abstraction fails in the context of persistent data, which unlike non-local program references, depends on the representation of the type. Barring some special system support for change (as discussed below), those instances would have to be modified or discarded, so as to conform to the (new) class specification. (The next section deals with this problem in detail.)

In the realm of database programming languages, it is very hard to escape the repercussions of change. Even the most traumatic type change cannot effect previously-compiled programs when those programs are run in isolation. Database applications, by definition, manipulate shared data. A change effecting the definition of a type that is represented in the database, renders old database program images obsolete. (These issues are discussed in more detail in Section 2.3.3 (p.10).) This thesis proposal will examine the issues related to the goal of keeping the (shared) data consistent and accessible in the presence of type evolution. While it may touch on some points relevant to the management of change in programs, that is not its focus. It begins by discussing the repercussions of type evolutions in the absence of any system support and proceeds

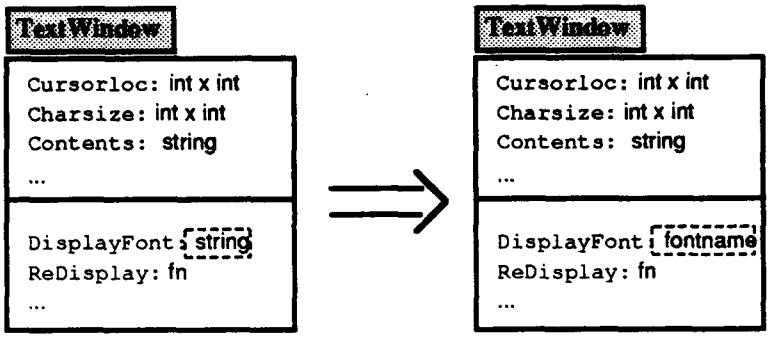


Figure 3: The change in domain of the public attribute `DisplayFont`, from `string` to `fontname`.

to discuss the various issues involved in providing integrated support. (Figure 4 (p.7) presents the breakdown of those design choices and issues, most of which are elaborated herein.)

2.2 Ad hoc Solutions

In a system lacking any evolutionary support, a modified type has no explicit relationship with its predecessor; from the perspective of the system, the type is totally new. The programmer has a policy decision to make: whether or not to convert the old instances into the new form. Since the database system lacks the means to assist in the conversion, any solution arrived at by the programmer to maintain compatibility between the existing instances and the new type will be done in an ad hoc manner, *i.e.*, using special code written particularly for this type evolution.

The argument promoting conversion proceeds as follows. The programmer obviously had a reason to redefine the type. Possibilities include: restructuring the representation to improve space and/or time efficiency, increasing functionality, deleting (now) irrelevant data fields, *etc.*

The conversion of existing instances will propagate those advantages to all instances. Without such a conversion, any application wishing to operate on the old and new forms of instances will have to design their implementations to accept and successfully manipulate both types.[5] This results in increased code size and programming complexity.

A programmer can attempt to reorganize the database to conform to her new type specification in one of two ways. She could work outside the confines of the database model, bringing down the system and manipulating the native representation of the data, or she could perform the conversion in the context of a database program, writing in the database programming language. In systems such as Statice and IDL [36, 9], database reorganization is only possible outside the context of the system. The rest of

[5]The effects of the change depend on both the degree of change and the modularity of the system.

- No system support (§2.2)
  - Reorganization within the system
  - Reorganization without the system
- System support (§2.3)
  - Type Change Specification (§2.3.1)
  - Restrictions on Type Changes (§2.3.2)
    - Restricted to compatible changes
    - Restricted to precalculated changes
    - Unrestricted
  - Consistency and Compatibility (§2.3.3)
    - Application dependencies
    - Method dependencies
    - Instance representations (§2.3.4)
      - Conversion
        - Specification
          - Canned conversion routines
          - Full inference
          - Programmer coding
        - Timing
          - Eager conversion (§2.3.4.1)
          - Lazy conversion (§2.3.4.2)
      - Emulation (§2.3.4.3)
        - Specification
          - Canned conversion routines
          - Full inference
          - Programmer coding

Figure 4: Type Evolution Support Decision Tree. The light nodes represent design choices that must be made, while the dark nodes represent issues that must be addressed for each choice. The section numbers refer to where these issues are discussed in detail.

this section, however, will discuss the problems associated with converting the existing instances of a changing type within the database system.

Database conversion following a type change is more than just generating one instance of the new type for every instance of the old. In database programming systems, objects are typically referenced by some type of universal identifier. This is in contrast to relational databases, where data is only accessible by value (*i.e.*, as the result of a query).

In order to preserve the references to the converted objects, any ad hoc routine must either (1) somehow preserve the identity of the object while changing its type, or (2) determine and alter all references to the affected instance, so that they point to the corresponding (new) instance. The ability to preserve the identity of an object while changing its type is supported by few database systems, IRIS [18] being the only such system familiar to the author.

In other systems however, a correct conversion program would need to accomplish three things: generate one new instance for every old instance of the changed type, delete all such old instances, and change all pointers to old instances to the corresponding new instance. In order to accomplish this, the programmer would need complete access to the database, so that the conversion routine could read and potentially alter all affected instances. In general, such capabilities are restricted to a few people under special conditions. Even more problematic is the case of external or proprietary databases, where the local owners are often not permitted to modify the database format at all [48].

It is clear that in the absense of some explicit system-based support for type evolution, the procedure is a painful and expensive one for programmers. The next section explores what type of support a database programming system can provide to ameliorate the situation.

2.3 System-supported Solutions

There are two distinct yet related issues that must be addressed by any system providing support for type evolution. The system must define a way for the user (*i.e.*, programmer) to specify her type change, and once in possession of this specification, the system must cope with the problem of making existing code and data compatible with it.

2.3.1 Specifying Changes to Type Specifications

Returning to our hypermedia example, consider the addition of a public attribute to out `TextWindow` class. Here, the programmer wishes to extend her system to support multiple font sizes. This is achieved by the addition of a new attribute, `Fontsize`. So far, it has been left unspecified exactly *how* this reimplementation is specified by the programmer. Basically, the change can be described either absolutely (similarly to how the class was defined originally) or relative to the existing class definition, in the

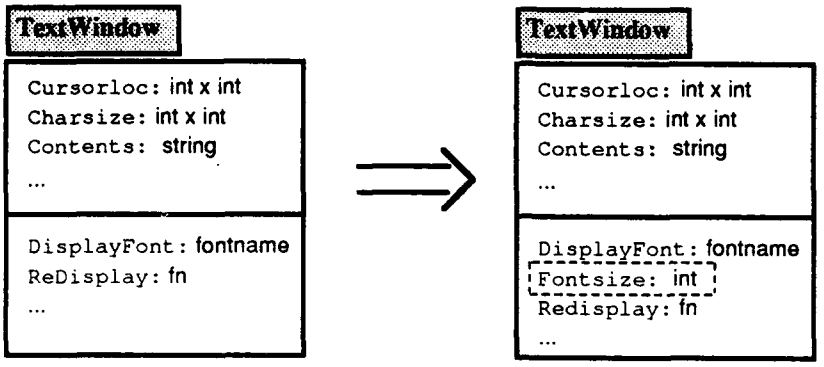


Figure 5: `TextWindow.AddAttribute("FontSize",integer,defaultfontsize)`: The result of the hypothetical evolution, which adds the `FontSize` attribute to the TextWindow class.

form of system routine calls on the class. (*cf.*, caption to Figure 5) Both schemes have advantages. Absolute redefinition is simpler from the point-of-view of the programmer, who would not need to learn a special "evolution language." However, if the system wishes to assist in reorganizing the database, these meta-linguistic routines provide explicit compatibility information to the system.

To illustrate a major advantage of relative vs. absolute specification, imagine a new change to our TextWindow example: the renaming of the `DisplayFont` attribute to `DefaultFont`. (*cf.*, Figure 6 (p.10)) All the system would be able to infer from a redefinition of `TextWindow` would be the deletion of the attribute `DisplayFont` and the addition of an attribute `DefaultFont`. However, renaming implies that the value of `DisplayFont` in all instances be preserved under the name of `DefaultFont` after the evolution.

The two approaches to specifying class evolutions are exemplified by the approaches of CLOS [45] and ORION [5, 28]. In the former, a `defclass` call on a existing class automatically converts all existing instances, by adding or removing slots as appropriate. In order to support changes that cannot be fully specified by redefinition (*e.g.*, renaming), the programmer has access to the (class-specific) method that creates new instances from old. The programmer can define functions to be invoked before or after the conversion routine, or rewrite the conversion routine altogether.

ORION supports evolution by providing the programmer with a list (*cf.*, Figure 1 (p.4)) of possible (that is, *supported*) evolutions she could perform. Associated with each selection is a reasonable conversion procedure, which cannot be modified by the programmer.

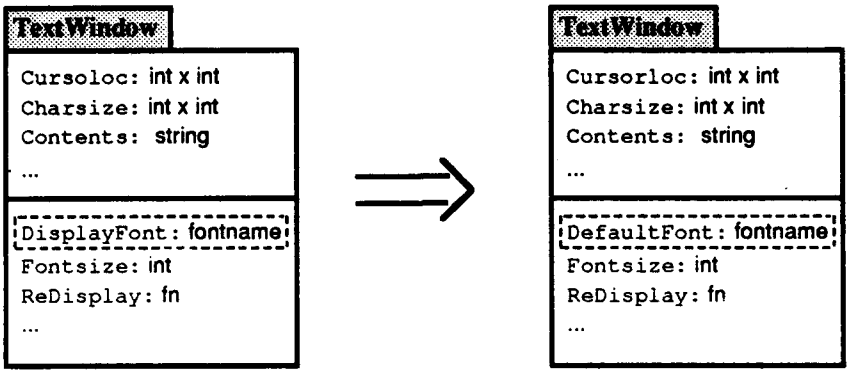


Figure 6: `TextWindow.RenameAttribute("DisplayFont","DefaultFont")`: The result of the hypothetical evolution , which renames the `DisplayFont` attribute in the TextWindow class to `DefaultFont`. This evolution is distinguished from the sequence of removing the `Font` attribute and adding the `Default` attribute by the preservation of the attribute values in all existing instances.

2.3.2 Type Change Restrictions

ORION, by listing the variety of class evolutions it supports, is restricting the range of schema evolutions possible in its database system. Although the rationale behind this decision is somewhat reasonable, it places unfortunate constraints on the programmer. Just because an evolution is hard or dangerous to support does not mean that it would never occur. (In fact, restricting the domain of an attribute field, one of the evolutions not supported by ORION, is not so uncommon.) By listing the range of "reasonable changes", ORION is essentially attempting to *anticipate* the changes the database programmer will need. The problem with this approach is that change is inherently unpredictable. Why can't type evolutions be *revolutionary*?

CLOS provides enough programming hooks to support the full range of changes. However, it does so by putting almost all (except for the task of locating the instances to convert) the responsibility on the programmer. ORION's list might not be compete, but it does enumerate a set of common evolutions, evolutions for which there are well-known, mostly undisputed, methods for preserving database consistency.

2.3.3 Database Consistency and Program Compatibility

In concert with how to allow the programmer to specify a type evolution, the system must address the issue of maintaining database consistency in the presence of change. As has already been illustrated, in addition to all the code references to the class, existing (persistent) instances are also dependent on its definition.

It is often possible to determine how to maintain consistency between the new class definition and its pre-existing instances. If evolution operations on the class definition

(*i.e.*, relative specification) were used, the direct specification of the evolution is often sufficient for the system to infer how to adjust the instance representation to conform to the revised type specification. For example, an addition of a class attribute can be propagated to all existing instances, so long as a default value for the new attribute is known. It is even earier to adjust for the deletion or renaming of an attribute. Often, however, additional information is required on the part of the programmer [31].

ORION has database consistency operators associated with each of its supported evolution operations. In fact, its designers support exactly those evolutions for which they found obvious and (relatively) non-controversial consistency operations. (Exactly *how* consistency is maintained is explained in later sections.) Some database systems are even more restrictive, disallowing any evolution on a class with instances [1]. Others only support the evolution of classes lacking subclasses, as that might complicated the database consistency maintenance scheme.[6]

Few systems make any effort to detect incompatibilities among existing methods or external applications. This is a hard problem in general, and the most existing systems do is to keep track of references to classes and class attributes and alert the programmer to *potential* incompatibilities resulting from the type evolution [59]. Perhaps a better avenue is to adjust the data to conform to the program instead of the other way around.

Before proceeding, a couple of definitions are in order:

Backward Compatibility is the provision for new programs (or, more generally, code objects) to be able to operate on old data. In our context, it would mean that instances created prior to the installation of the type change remain accessible through the revised type.

Forward Compatibility is the provision for programs compiled prior to the change to be able to operate on new data, *i.e.*, to be able to manipulate data created after the type change as if it had been created earlier. (Note that the prior compilation stipulation makes our definition more restrictive than the standard.)

Some evolutions are inherently backward or forward compatible. Changes which only remove information from a class are naturally backward compatible, while changes only adding information are forward compatible. Interface-preserving operations should be both forward and backward compatible. In general, however, evolutions are not program compatible, and systems wishing to preserve compatibility across evolutions must do extra work.

2.3.4 Implementing Database Consistency: Conversion vs. Emulation

This section contrasts the various implementation strategies for maintaining database consistency. The primary distinguishing factor is whether to actually adjust the data

[6] The problem is with multiple inheritance, I suspect.

representation of the objects in the database (*i.e.*, conversion) or to install code to mediate among the interfaces (*i.e.*, emulation).

The general merits of data conversion were discussed previously. (*cf.*, Section 2.2 (p.6).) Integrated conversion facilities provide a built-in procedure to iterate over all instances, often inaccessible to the programmer. This can solve the security and identity problems attributable to ad hoc conversions.

2.3.4.1 Eager Conversion

In eager conversion, the transformation of the instances is done simultaneously with the specification of the change.

The scheme has some significant advantages and disadvantages. The primary feature is that all the expense of the conversion is spent at evolution time, so no additional runtime support is required. However, the disadvantages are serious. The cost of an eager database reorganization is proportional to the number of relevant instances in the database (sometimes even proportional to the size of the *total* database). During the reorganization process, access to the database is monopolized by the conversion routine. This is a serious deficiency in situations where high availability of the database is of significant importance. As a result of the conversion, all existing database applications must be (in the worst case) rewritten to conform to the new type definition. This expense is notable if many copies of the applications exist, if the creators of the older programs are no longer around to update them, *etc.* An aspect of an older issue is a problem here as well. In a database supporting restricted access permissions on data, the system must circumvent the security provisions in order to propogate the conversion. In order to successfully convert each and every instance in the database, the system will have to access all relevant instances, even those the programmer may not have access to herself.

The most well-known system employing eager conversion is GemStone, which converts its instances in concert with its garbage collection of the persistent store [10].

2.3.4.2 Lazy Conversion

Instead of converting all instances at type evolution time, the lazy conversion scheme defers the conversion of a relevant instance until that object is accessed by an application program. With respect to specifying how to perform the conversion, this procedure is similar to the eager conversion scheme.

Notable advantages of lazy over eager conversion are:

(Access to data) The conversion is performed on behalf of the owner (or ward) of the data, so the security problem associated with eager conversion is avoided. Also, in distributed database systems, lazy conversion is easier to perform than eager conversion, as a global synchronization on a type and *all* of its instances is not required.

(Reduced denial-of-service time) Only the installation of the new type requires serial access to the database. The conversion of individual instances can be done in association with other operations.

All is not rosy, however, as lazy conversion requires more system support than eager conversion. One complication associated with this scheme is whether or not to perform the conversion when the old object is not being modified; the conversion of objects at read time would make those operations much more expensive. Also, unlike the eager method, this procedure depends on runtime system support [7, 10].

Although the CLOS specification allows systems to implement instance conversion either lazily or eagerly, it should be noted that all known CLOS implementations (PCL, Lucid, *etc.*) convert their instances lazily [17, 27, 13].

2.3.4.3 Emulation

Rather than actually convert old instances, under the emulation scheme, all interaction with older objects is via a set of filters, which support the new-style operations on the old-style data format. All of the information associated with the old type is retained, the filter functions masking their presence from the programmer. It is also possible to allow old programs to operate on new type definitions, by writing filter routines that make new instances appear as old ones.

Emulation is considerably less efficient than the conversion schemes, as the cost of emulation is paid every time an object is referenced.

Support for emulation instead of conversion changes the way programmers must think about change. Although the actual object translation might be deferred, conversion forces the programmer to consider her versions in a linear order, `Version 1` instances are converted into `Version 2` instances which are then converted into `Version 3` instances, /etc. With emulation, all the past versions of a type remain indefinitely, so the programmer must consider how the new version of the type relates to all past versions, not just the most recent one. This likely results in an increase in complexity, as the programmer might have to define filter functions for every feature of the current type for every possible past type-version.[7] Some system assistance, similar to the automatic conversion routines supplied by ORION, might be possible, but the multiplicity of active type-versions is likely to complicate matters, and this area has not been explored.

For all the additional expense, emulation delivers additional functionality over the conversion schemes. In situations where the database cannot be converted (read-only or proprietary databases, for example), emulation is the only alternative. Zdonik, the primary supporter of the emulation scheme in the literature, identifies three other reasons why conversion is not always an option: Often it is impossible to define a conversion

[7] **Note that the author has yet to actually define the term. It is hoped, that for time being, its meaning can be deduced from context.**

routine because of a lack of information in the old object; often the conversion will result in a loss of information[8]

The most important practical advantage of this scheme is the inherent support for forward compatibility, allowing old applications to continue to function after the type evolution. This is particularly useful in cases where such applications cannot be recompiled, or if copies of the application have been distributed [42].[9]

The ENCORE research system [26] is the only available system that supports emulation. The old AVANCE research system [6, 7] supported it as well.

It is also worthy of note that relational database systems essentially provide emulation support through *relational views*[10] [14]. Under such a scheme, the reformed table is give a new name, and the original table is replaced with a relational view, which emulates the original relation through the use of relational `SELECT` and `JOIN` operations.

2.4 Mid-proposal Review and Conclusions

So far, this proposal has presented the current state of technology with respect to type evolution support, particularly in the context of object-oriented databases. It has made the following points:

- Type evolution is inherent to programming in general, and characteristic of design applications in particular.
- Object-oriented databases, created specifically to support design applications, need to address type evolution.
- Database systems in general would benefit from database consistency support.
- Non-integrated (ad hoc) solutions offer limited functionality and some considerable disadvantages (such as downtime).
- With respect to specifying change, an evolution "shopping list" is convenient, but as change is unpredictable and often unavoidable, the system should encourage the programmer by providing as much support as possible, rather than placing restrictions on the range of possible evolutions.

[8] There is an implicit and explicit loss of information attributable to some conversions. If the type evolution resulting in the deletion of an attribute, than an explicit loss of information might occur. The implicit loss is related to the implicit age of the object owing to its type-version. After conversion, the object will be indistinguishable from newly-create ones [48].

[9] Depending on the modularity and linking properties of the programming language, a relinking might be necessary to allow old applications to operate on new object instances. However, unlike previous schemes, *no reprogramming is necessary.*

[10] Actually, it is a combination of relational views and runtime binding of programs to table that makes this work in RDBMSs.

- Emulation is inherent inefficient, but should not be casually dismissed by an implementation supporting database consistency and program compatibility, as it offers considerable functionality over strict conversion schemes.

There remains a considerable amount of space to explore between eager conversion and pure emulation, representing implementations which return some of the performance of conversion schemes while retaining some of the desirable functionality of emulation. In some situations, programmers would be undoubtedly willing to sacrifice some performance to realize additional versatility in their database applications.

In the next section, I motivate mixed approaches to database consistency support and type evolutions specification.

3 Towards a More General and Effective Conversion Strategy

The previous section presented the state of technology for type evolution in existing modern database systems. This section proceeds to describe what features *should* be supported by new database systems, and why.

3.1 Motivations for Hybrid Strategies

Three general strategies for supporting database consistency following a type evolution were previously discussed (2.3.4 (p.11)). Since none of these three is ideal in all circumstances, it is unfortunate that none of the new OODB systems support more than one of these schemes. The programmer is given no opportunity to select the database consistency scheme best suited to her needs.

Even is such a possibility were available, however, the programmer would still be limited to only three choices. In reality, there are many possible consistency schemes that could result in better database performance, if only they could be made available to the database programmer. For example:

- Imagine a very large and active database with a small set of "hot" entries (*i.e.*, items that are accessed very frequently), a relatively large set of "cold" entries that are accessed very infrequently, and a still larger collection of entries that will remain unaccessed for the foreseeable future ("frozen objects?"). Let us assume further that the recompilation of applications is not an obstacle, making backward compatibility a nonissue.[11] Clearly none of the schemes is optimal. Eager conversion is too impractical (the database is too large), emulation would result in significant performance penalty on the hot items, and lazy conversion would spend considerable time converting the cold entries, when emulation in those instances might be more economical. It would seem that the following hybrid strategy would perform better than any of the three basic schemes.

  > *In an attempt to improve runtime performance, convert only the hot entries, and emulate the cold ones. This can be achieved by recording the number of times an item is accessed, converting it only when its access count exceeds a certain threshold.*

  The value of the threshold depends on how expensive a conversion is as compared to an emulation operation. Note that this mixed conversion/emulation scheme is a generalization of the lazy conversion scheme; setting the threshold for conversion

[11] This example is reminiscent of an electrified library, where all reference materials are available online.

to 0 would implement lazy conversion exactly.[12] This scheme could be extended further to support different thresholds for different types (which presumably would have different emulation-to-conversion cost ratios.)

- Suppose that you wish to extend a data type by adding another component (*e.g.*, a citizenship field to an employment record schema). Rather than convert all the existing instances to the new format (and inserting `citizenship:UNKNOWN` into their representations), it would be more space efficient (and no less time efficient) to emulate the citizenship component with a special function, until such time (if ever) as the citizenship of the employee can be determined and entered. Here, one would like to emulate the new type-version (for space efficiency) as long as possible.

- Suppose your database system supports emulation only. After you have evolved a new type-version, and execute with it for a while, you change your mind and wish to "back out" of your upgrade. Within the emulation scheme, you could make the previous type-version the *current* one, so that all new instances and applications would resolve against the that definition. However, there is still the problem of your database being populated with some number of instances of the experimental type-version. Given that there are no (longer any) applications expecting that version, it would be preferable if you could convert those instances into "old-new" ones, and *rollback* your change.

3.2 Desiderata

I have argued that type evolution support systems should be more flexible than those provided by current implementations. Here is a list of the features I consider important for an implementation of a multi-purpose type evolution support system:

1. Old applications should be able to operate on new data, and new applications should be able to operate on old data. (**Full Compatibility**)

2. Emulation should be available as an option, since there are times when emulation is the only reasonable solution.

3. Conversion should be supported, because there are times when conversion is most appropriate.

4. The system should provide assistance in the generation of conversion and/or emulation routines.

[12] Maier *et al.* see emulation (they call it screening) and lazy conversion as variations of the same process. While lazy conversion performs the actual conversion on first access, emulation defers the conversion operation indefinitely.[10]

5. The system should be able to implement hybrid support strategies, such as those described in Section 3.1.

6. All of the features should be optional, used at the discretion of the type implementor.

I envision a type evolution support system that is customizable and extensible. Just as it is not possible for a type designer/programmer to anticipate the range of applications of her work, it is not possible to anticipate all the possible forms of changes that could be made as part of the evolution of a data type specification and implementation. Rather, the support system should be able to accept conversion and backward compatibility instructions from the programmer, and proceed to implement them as efficiently as is possible. It should also be possible for the programmer to alter the support strategy even after it has been installed,[13] as more infomation about the costs is available.

The features of conversion, emulation, and computer assistance have been discussed previously. The remainder of this section will explain the other issues in more detail.

3.2.1 Full Compatibility

Full compatibility encompasses both forward and backward compatibility, as defined earlier (p.11). It is worthwhile to elaborate upon the strict definition of backward compatibility, which stipulates that old programs must continue to execute correctly, unchanged, on new instances. Since the programmer of the type is often not the sole user of the type, the fact that a change occurred must otherwise be communicated to all application writers. If copies of the applications have been widely disseminated, then we have a software distribution problem. Moreover, applications often come "shrink-wrapped", making recompilation impossible. The stricter definition of backward compatibility avoids both of these problems. An important feature of the emulation support scheme for type evolution is basically the principal of full compatibility [48]. Full compatibility is not a quality exclusive to emulation, however. It is just that emulation lends itself well to it. For instance, given only the ability to convert between instances of two type-versions, one would be possible to implement full compatibility, the representation of the object alternating between the two type-versions whenver necessary. Clearly this strategy is far from efficient. A method for developing more appropriate strategies will be illustrated later. (*cf.*, Section 3.2.3.)

3.2.2 Emulation and Lazy Conversion

The existence of support for both emulation and lazy conversion in one system allows for a number of interesting possibilities. Even if you wish to provide backward compatibility, you are not necessarily required to defer the (imminent) conversion of old

[13] **evolutionary evolution strategies!**

instances indefinitely. It is quite legitimate to convert the old objects and perform the emulation *backwards* for old applications, as opposed to emulating the old objects *forward* for the benefit of new applications. It would seem like the former option would be preferred if old objects (either in general or a subset of them in particular) were accessed more frequently by new programs than by old.

Another potential optimization is a generalization of the "version rollback" scheme, mentioned in Section 3.1 (p.17). In situations where there are no remaining applications expecting a particular type-version, and where the instances of that version could be converted to an active version without loss of information, conversion of those instances would be potentially useful.

3.2.3 Hybrid Strategies

A set of possibly useful hybrid strategies have already been alluded to. A simple, yet potentially useful one that was discussed anecdotally earlier (*cf.*, Section 3.1 (p.16)) extended lazy conversion by parameterizing the number of access operations required to be performed on an object before it would be converted. It was also casually mentioned that it could be further parameterized on the type (perhaps even on the type-version) of the instance itself. An interesting extension of this strategy (with some theoretical underpinnings) is presented in Appendix B.

Tuning performance by playing with the time before which an object can be converted can only take you so far. In general, the granularity of the options (either the object is an instance of the old type-version or it is an instance of the new) is too large. Better performance is likely possible for some mix of applications if it were possible to somehow *merge* the specifications of the two (or more) type-versions, supporting both backward and forward compatibility at a (minimized) cost to both old and new clients.

In fact, we do not have to wish for this possibility for too long as there exists a technology that addresses this very issue. That technology is *view-based abstraction*[14]. Views[21, 23] are a mechanism for supporting multiple external specifications on an abstract data type. The resulting system supports instances that may be operated upon by different applications using different type specifications, but which are all operating on a common instance, whose concrete implementation represents all the **facets** (*i.e.*, views of an instance) of the object.

A significant class of hybrid evolution support strategies can thus be represented as a view implementation problem, with full compatibility supported though various views of a type (one for each active type-version). Precise details on how views can assist with type evolution are the focus of Section 4.

[14] These views are distinguished from the relational views referred to earlier.

3.2.4 Opting Out, Programmability, and Extensibility

As mentioned before, I hardly expect to be able to anticipate the range of reasonable conversion strategies, so I wish the system to provide a considerable amount of flags and hooks to allow the programmer to tailor the system to her needs as much as possible. Flags will allow the programmer to turn off certain (expensive) features that she has decided are unnecessary for her situation, while hooks will provide the programmer with ultimate control over the conversion/emulation process.

3.3 Evaluating Evolution Support Strategies

In Section 3.1, I argued that a hybrid strategy will often perform better than those that have been proposed and implemented to date. My argument, however, was both anecdotal and informal. As part of my thesis work, I propose to develop a (more) formal framework within which I will be able to evaluate the various evolution support strategies. This framework will have to model database object access patterns (variety and frequency), as well as provide a method for comparing the costs of the applicable compatibility strategies (total conversion of instance, total emulation, *etc.*.).

As a first attempt at a model, I have formalized the economics of the three basic evolution support strategies. The result is described in Appendix A (p.39).

Once I have an analytic method for evaluating evolution support schemes, I will be able to quantitatively compare the tradeoffs involved in choosing one strategy over another. This applies not only to the three basic schemes, but to the other strategies that I intend to develop in the course of my research.

4 Type Views as Abstraction of Change

Earlier (*cf.*, Section 3.2.3 (p.19)), I referred to views as an existing technology that could be used to implement hybrid conversion strategies. This section explains what views are and elaborates on how they can be applied to the problem of type evolution.

4.1 Brief Historical Interlude

Views were developed as a way to design a system that supports multiple methods of access to a common body of data. These access methods are distinct and usually motivated by different applications. For instance, in the design of a compiler, the various modules (parser, typechecker, flow analyser, *etc.*) manipulate shared data, mainly, the source program. Each module, however, views the source differently. The parser views it as a sequence of lexical tokens; the typechecker — as a graph of language-level constructs.

As is always the case when more than one agent influences the design of an entity, independent specification is followed by a negotiation phase where all the factors are considered and the various features of the entity are merged together to establish a common design accomodating to all. In general programming, this would require establishing two things: a common data representation and a common type signature. With views, however, although there is a common data representation, the type signature, the interface to the data, is application- (or **view-**) specific. Once the merged representation (called the **merged view**) has been established, the various methods associated with each of the views must be redefined. The individual applications do not need to be altered, since their view of the data is unchanged.

For an implementation design of views, readers are referred to the work on Janus [23], by Habermann *et al.*.

4.2 An Abstraction for Compatibility

Views were developed as a way to cope with multiple interfaces to common data. The full compatibility functionality touted in this proposal fits very well into a "views" framework, with each view corresponding to a supported version of the type. In this way, a view-based type system can implement the versioning of types.

By way of example, it is certainly not hard to imagine a type evolution where the representation and signature of the type **`ComplexNumber`** is changed from rectangular to polar coordinates. In such a situation, program compatibility can be retained by defining the type as a merged type, supporting both rectangular and polar views.

A view-based type system localizes the problem of specifying compatibility. In developing a merged view and by defining procedures for the signatures of all the views (*i.e.*, the supported type-versions), the type designer has defined compatibility properties between the various active versions.

The Janus manual identifies four storage models for merged view representations: **disjoint, shared, derived**, and **"Anything Goes"**. These models correspond roughly to the various compatibility strategies described in the context of schema evolution, as described in the following table:

Storage Model	Merged view is represented using...	Roughly corresponding evolution compatibility strategy
Disjoint	...the disjoint union of the fields of the component types.	conversion[15]
Shared	...the common fields of the component types.	(none)
Derived	...the representation of one of the original types. Values for the others are calculated dynamically.	emulation
"Anything Goes"	...some other representation. There is no direct correspondence between the merged representation and the representation of the original types.	hybrid strategies

Consider the implementation of the attribution addition evolution example (*cf.*, Section 2.3.1 (p.8)). Figure 7 (p.23) provides an implementation of that evolution under Janus' views.

It is within the context of the "Anything Goes" model that the programmer has the opportunity to optimize her representation and view methods to the body of active applications and type-versions. The fine granularity referred to wistfully earlier (*cf.*, Section 3.2.3 (p.19)) is available in this context, allowing for optimizations at the level of the constituent items of the representation, as opposed to at the instance level.

4.3 An Abstraction for Conversion

While views as described are a useful abstraction for thinking about and programming version compatibility, they do not address the problem of conversion of existing instances.

Let us return to the attribute addition example implemented in Figure 7. While supporting both the old and new versions of `TextWindow`, the merged representation is only consistent with instances created after the evolution. As it stands, the merged class only provides forward compatibility, allowing both new and old programs to manipulate

[15] If we could imagine a system attempting to support full compatibility, but which only has conversion routines (as opposed to emulation filters) available, a reasonable implementation would be to maintain multiple copies of the object, one for each type-version, and to reinitialize each following the modification of one. This is not strictly true, which is why this is only a rough analogy.

```
merged class TextWindow is
        interface
             facet TextWindow'v1;
             facet TextWindow'v2;
        end interface;

        implementation
             field Cursorloc:  Int x Int;
             field Charsize:  Int x Int;
             field Contents:  String;
             ...
             field Font:  fontname;
             field Fontsize:  Int;
             ...

             facet TextWindow'v1 is
                  operation Redisplay() is ...
                  operation Get_Font() is ...
                  operation Set_Font(fn:  fontname) is ...
                  ...
             end TextWindow'v1;

             facet TextWindow'v2 is
                  operation Redisplay() is ...
                  operation Get_Font() is ...
                  operation Set_Font(fn:  fontname) is...
                  operation Get_Fontsize() is ...
                  operation Set_Fontsize(i:  int) is...
                  ...
             end TextWindow'v2;

        end implementation
end TextWindow
```

Figure 7: The attribute addition example (*cf.*, Section 2.3.1 (p.8)) implemented as a Janus view definition. The merged class is implemented using the representation of the new **TextWindow** specification, and the procedures related to the **'v1** and **'v2** views just pass the values across.

the new multi-faceted instances.[16] It fails, however, to explain how instances existing prior to the evolution, which lack a value for the `Fontsize` field, can be manipulated by new programs, which expect instances with a `'v2` facet.

This shortcoming will have to be solved before this view-based approach can be used as part of a type evolution support system. Here is an approach currently under consideration that I believe will lead to a good solution.

> First, we associate a `Consistent` flag with each facet of a (merged) object. A set `Consistent` flag means that the object's fields accessible via this facet's operations are consistent with the state of the object as a whole. When an object is created, only one of its facet is consistent — the one corresponding to the view used to initialize the object. When a program accesses an object via an inconsistent facet, the object must first be made consistent with that view, before the access can proceed. The method used to make a facet consistent is similar to the method used by the old lazy conversion strategy to convert from one type-version to another. However, since the view implementation supports sharing among the various facets, the conversion routines here should be more efficient.
>
> With the use of these flags, there are, in effect, $2^v - 1$ *virtual* representations (v = the number of active type-versions) of the object, corresponding to the 2^v possible states of the v `Consistent` flags. (The state where all the flags are clear has no meaning.)

4.4 Problems Inherited from the View Model

There remain a number of issues, related to the database consistency problem and the views model as a whole, which must be addressed as part of the thesis research. Some of these were acknowledged by Habermann *et al.* [23, pp. 74–82].

Subclassing and Inheritance. The scheme as described did not support subclassing or inheritance. Since most object-oriented language systems provide these mechanisms, some resolution in this area is required.

N-Way Merging. The compatibility problem has been focused on the case of two type-versions. (*cf.*, Section 13) The abstraction will have to be able to handle the case where other versions are added.

Concurrent Evolutions / Non-linear Versions. The version space of type definitions throughout this proposal has always been linear (*i.e.*, successive evolutions on a type specification). It would be interesting to see if this approach could be reasonably extended to support version hierarchies.

[16] This is possible because old and new programs can manipulate the object via the `'v1` or `'v2` view, as appropriate.

5 Related Work

There are a number of other research threads that relate to this work.

Balzer's stuff

Balzer [4] has done some work on the enhancement of Knowledge Representations, which I tend to perceive as AI databases. Balzer was primarily concerned with what he calls *structural enhancement*, modifications which involve the restructuring of the (database) format.

Object Versions

The versioning mechanism on objects can be easily extended to types as well, especially in an object-oriented framework, where the classes are objects. AVANCE [6] took that direction. However, the issue of version management is different in focus than the main problem addressed in this thesis, which is the conversion of old instances.

Views

Application of views technology will obviously be an important aspect of my research. I expect to draw upon the body of work that has been done or is in progress in that field.

Garbage Collection Strategies

Incremental and stop-and-copy garbage collectors have been around for a while, and their implementations might help me to develop conversion strategies. The garbage collection process has a number of similarities with database conversion. Consider stop-and-copy garbage collection, where the collector scans through the entire heap (which is a form of short-lived database) following pointer references. This scan phase is similar to the procedure an eager conversion program would have to follow if it were attempting to update objects in a similar environment. The difference between the (stop-and-copy) collector and the (eager) conversion program is what they do once they have located an object. The collector copies it to a new heap partition unaltered, while the conversion routine, in certain instances performs its conversion.

This analogy holds up across implementations. Consider the similarities between lazy conversion and an incremental garbage collector, which spreads the process of object migration out over the course of the computation.

For this reason, I believe that the field of garbage collection schemes and implementations has applications in the world of object conversion and database reorganization.

Program Restructuring

There are some similarities between the problem of the automatic restructuring of programs after a module or procedure modification. [22] The analogy works if you consider the specification of a module or procedure as its schema (or type), and the references (uses) to the module or procedure as an instance.

Functional Databases

In functional databases, all previous states of types and instances are preserved forever, but eventually moved to slower storage media for economical reasons. Their perspective on change is a valuable one. [25]

Formal Module Manipulation

Nord *et al.* [37], for example, are using formal techniques to develop a mechanized way of merging different representations of modules. This technology would be ideally suited to the process of finding a type representation that could support the functionality of emulation, but with costs approaching that of conversion. In our view-based model, this technology could automatically develop the merged type representation.

6 Research Plan

In the preparation for this proposal, I have already begun work on the following relevant items:

- Compare, contrast and evaluate various existing approaches to support for schema evolution.
- Develop performance evaluation model for evolution support strategies.
- Search for real world information regarding importance of various types of evolutions. Also acquire access to existing (research) systems incorporating persistence (*e.g.*, . Alexandria [46], ASCENT [38], Garnet [19]).

In the course of my research I will have to accomplish the following additional things:

- Extend model of views:
  1. So they can better function as abstraction of conversion. (§4.3)
  2. To cope with inheritence, and other object-oriented concepts. (§4.4)
  3. To cope with concurrent evolutions (§4.4)
- Develop propotype system on top of a real database system. This will involve building an *Evolution Management Layer* in which the notion of type-versions and views will be supported.
- Experiment with different hybrid strategies (*i.e.*, merged (view) representations), supported by the weakening of consistency constraint between views (§4.3), and motivated by competitive algorithms, (user-provided and inferred) access models, *etc.*.
- Using performance evaluation model, identify system and application characteristics and constraints which would tend to favor certain evolution strategies over others. To accomplish this I will need access to some realistic database applications (see above).
- Experiment with system and various strategies in a number of domains, *e.g.*, design databases, traditional database application, hypermedia database. I plan to use toy (and possible real) versions of the database applications I acquire access to.

In the course of my research the following issues might come up, but I hope to be able to keep from addressing them:

- **Type systems and type evolutions.** One can consider the relationship between type-versions as a hierarchy orthogonal to the inheritance graph, with *can-simulate* arcs connecting them, instead of *is-a* or *is-refinement-of* arcs. Simulation relationships can possibly exist between objects that are not strictly evolutions (*e.g.*, a box can function as a chair, under certain conditions), making the hierarchy resemble a partial lattice more than a graph.
- **Evolution and Transactions.** What are the transaction semantics of a schema evolution?
- **Evolution and Database Queries.** How can the representation of objects be adjusted over time, without unduly penalizing the efficiency and correctness of database queries?

Figure 8 illustrates the various dependencies between the research goals while figure 9 associates the research goals to a timetable.

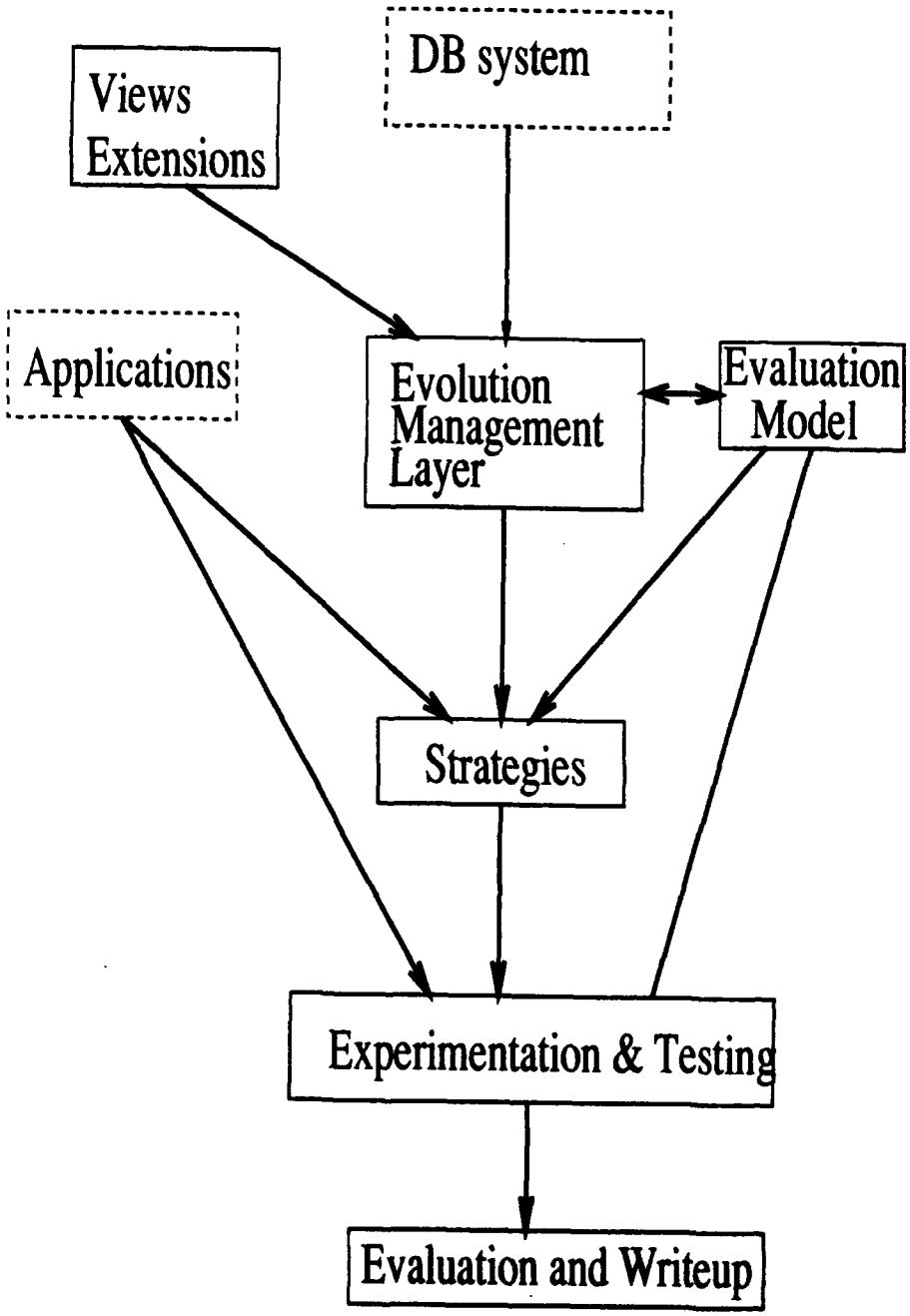


Figure 8: Dependency Diagram for Research Plan. The dashed boxes represent search tasks, while the solid boxes are development tasks.

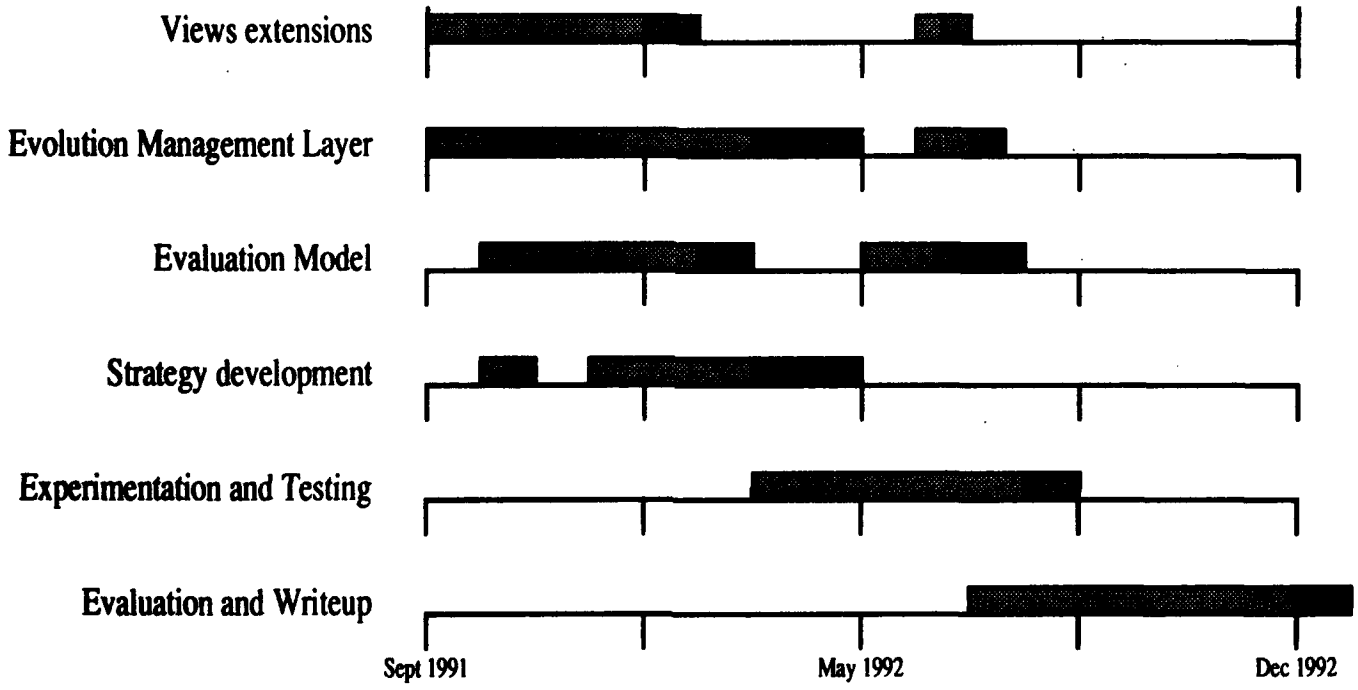


Figure 9: Research Goal Timeline

7 Contribution

The continued integration of computers into our society has been hailed as part of the *March of Progress*. It is time that the developers of computer-based systems examined the costs of change in their own contexts. This thesis will explore but one aspect of the ramifications of change within computer systems. As new or modified applications necessitate a change in the form of the persistent data, how can one best continue and maintain its substance?

The research plan that I have outlined here focuses on the evolution of types within an integrated, typed, database system. Hopefully, its conclusions will illustrate the following points:

- It is feasible and reasonable to have a DBMS that has the ability to support multiple applications and multiple versions of the same application, all operating on the same data. One of the functions of persistence is to support sharing; my work will extend the range of such support.
- Such functionality can be added without significant performance degradation.
- An extension of the principle of modular system design. In situations where no information has been lost in the database reorganization, old applications can continue to operate correctly, unobstructed by the type change.

I further to make secondary contributions towards these goals:

- Partial automation of programming to support database conversion and program compatibility benefits the system by reducing programming complexity, encouraging reliability, and minimizing the costs of the labor-intensive process of data conversion.
- The described mechanism might provide a course of action for coping with the *Software Release Problem*, *i.e.*, how to deal with the variety of source files in the context of multiple versions of a software product.
- The described mechanisms could help in the design of support for long-lived transactions, as programmers would be able to make changes to instantiated type definitions without interfering with other programmers.

References

[1] Albano, A., Cardelli, L., and Orsini, R. *Galileo: A Strongly-Typed, Interactive Conceptual Language.* **ACM Transactions on Database Systems**, vol. 10 (1985), pp. 230–260.

[2] Atkinson, M. P. and Buneman, O. P. *Types and Persistence in Database Programming Languages.* **ACM Computing Surveys**, vol. 19 (1987), pp. 105–190.

[3] Atkinson, M., Bailey, P., Chisolm, K., Cockshott, W., and Morrison, R. *An Approach to Persistent Programming.* **Computer Journal**, vol. 26 (1983), pp. 360–365.

[4] Balzer, R. *Automated Enhancement of Knowledge Representations.* in: **Proceedings of the Ninth International Joint Conference on Artificial Intelligence.** 1985, pp. 203–207.

[5] Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases.* in: **Proceedings of the SIGMOD International Conference on Management of Data**, edited by U. Dayal and I. Traiger. San Francisco, CA, 1987.

[6] Björnerstedt, A. and Britts, S. *AVANCE: An Object Management System.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA).** San Diego, CA, 1988, pp. 206–221.

[7] Björnerstedt, A. and Hultén, C. *Version Control in an Object-Oriented Architecture.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[8] Bloom, T. and Zdonik, S. *Issues in the Design on Object-Oriented Database Programming Languages.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA).** Orlando, FL, 1987, pp. 441–451.

[9] Borison, E. *Personal Communication.*

[10] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. *The GemStone Data Management System.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[11] Casais, E. *Managing Class Evolution in Object-Oriented Systems.* in: **Object Management**, edited by D. C. Tsichritzis. Centre Universitaire d'Informatique, Geneva, 1990, pp. 133–195.

[12] Clamen, S. M. *Data Persistence in Programming Languages - A Survey.* Technical Report, no. CMU-CS-91-155, Cambridge, MA, Pittsburgh, PA, May 1991.

[13] `comp.lang.clos` newsgroup discussion. 1991.

[14] Date, C. **An Introduction to Database Systems.** 4th. vol. 1, Addison-Wesley, Reading, MA, 1981.

[15] Detlefs, D., Herlihy, M. P., and Wing, J. M. *Inheritance of Synchronization/Recovery Properties in Avalon/C++.* **Computer**, vol. 21 (1988).

[16] Deux, O. and *et. al. The Story of O_2.* **IEEE Transactions on Knowledge and Data Engineering**, vol. 2 (1990), pp. 91–108.

[17] Dussud, P. *An Implementation of CLOS for the Explorer Family.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA).** New Orleans, LA, 1989, pp. 215–221.

[18] et. al., D. F. *Overview of the Iris DBMS.* in: **Object-Oriented Concepts, Databases and Applications**, edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989, pp. 219–250.

[19] *et al.*, B. A. M. *The Garnet Toolkit Reference Manuals: Support for Highly Interactive, Graphical User Interfaces in Lisp.* Technical Report, no. CMU-CS-90-117, Cambridge, MA, Pittsburgh, PA, March 1990.

[20] Garlan, D., Kaiser, G., and Notkin, D. *On the Criteria To Be Used in Composing Tools into Systems.* Technical Report, no. TR 88-08-09, University of Washington. Seattle, WA 98185, August 1988.

[21] Garlan, D. B. *Views for Tools in Integrated Environments.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 1987. *Available as Technical Report CMU-CS-TR-147.*

[22] Griswold, W. G. and Notkin, D. *Program Restructuring to Aid Software Maintenance.* Technical Report, no. 90-08-05, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195 USA, September 1990.

[23] Habermann, A. N. *et. al.. Programming with Views.* no. CMU-CS-TR-177, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, 1988.

[24] Heiler, S., Dayal, U., Orenstein, J., and Radke-Sproull, S. *An Object-Oriented Approach to Data Management: Why Design Databases Need It.* in: **Proceedings of the 14th ACM/IEEE Design Automation Conference.** 1987, pp. 335–340.

[25] Heytens, M. L. and Nikhil, R. S. *GESTALT: An Expressive Database Programming System.* **SIGMOD Record**, vol. 18 (1988), pp. 54–67.

[26] Hornick, M. F. and Zdonik, S. B. *A Shared, Segmented Memory System for an Object-Oriented Database.* **ACM Transactions on Office Information Systems**, vol. 5 (1987), pp. 70–95.

[27] Kiczales, G. and Rodriguez, L. *Efficient Method Dispatch in PCL.* in: **Proceedings of the ACM Conference on Lisp and Functional Programming**. Nice, France, 1990. pp. 99–105.

[28] Kim, W., Garza, J., Ballou, N., and Woelk, D. *Architecture of the ORION next-generation database system.* **IEEE Transactions on Knowledge and Data Engineering**, vol. 2 (1990), pp. 109–24.

[29] Kim, W., Ballou, N., Chou, H.-T., Garza, J., Woelk, D., and Banerjee, J. *Integrating an Object-Oriented Programming System with a Database System.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)**. San Diego, CA, 1988, pp. 142–152.

[30] **Object-Oriented Concepts, Databases and Applications**. edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[31] Lerner, B. S. and Habermann, A. N. *Beyond Schema Evolution to Database Reorganization.* in: **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA) and Proccedings of the European Conference on Object-Oriented Programming (ECOOP)**. Ottawa, Canada, 1990, pp. 67–76.

[32] Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W. *Argus Reference Manual.* no. TR-400, MIT Laboratory for Computer Science, Cambridge, MA, November 1987.

[33] Maier, D. and Stein, J. *Development and Implementation of an Object-Oriented DBMS.* **MIT Press Series in Computer Systems**, MIT Press, Cambridge, MA, 1987, pp. 355–392.

[34] McCleod, D. and King, R. *Semantics Database Models.* in: **Principles of Database Design**, edited by S. Yao. Prentice-Hall, 1984.

[35] Monk, S. and Sommerville, I. *A Model for Versioning Classes in Object-Oriented Databases.* Internal Report, no. SE-91-07, Computing Department, Lancaster University, Lancaster, UK, July 1991.

[36] Nestor, J. R., Newcomer, J. M., Giannini, P., and Stone, D. L. **IDL: The Language and Its Implementation. Prentice Hall Software Series**, 1990.

[37] Nord, R. L., Lee, P., and Scherlis, W. L. *Formal Manipulation of Modular Software Systems.* in: **Proceedings of the ACM SIGSOFT International Workship on Formal Methods in Software Development**, The Association of Computing Machinery. ACM Press, 1990, pp. 90–99.

[38] Piela, P. C., Epperly, T. G., Westerberg, K. M., and Westerberg, A. W. *ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language.* **Computers and Chemical Engineering**, vol. 15 (1991), pp. 53–72.

[39] **Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA).** San Diego, CA. 1988.

[40] Schmidt, J. W. *Some High Level Language Constructs for Data of Type Relation.* **ACM Transactions on Database Systems**, vol. 2 (1977), pp. 247–261.

[41] Shriver, B. D. and Wegner, P. **Research Directions in Object-Oriented Programming. MIT Press Series in Computer Systems**, MIT Press, Cambridge, MA, 1987, 585 pp.

[42] Skarra, A. H. and Zdonik, S. B. *Type Evolution in an Object-Oriented Database.* **MIT Press Series in Computer Systems**, MIT Press, Cambridge, MA, 1987, pp. 393–415. *An early version of this paper appears in the OOPSLA '86 proceedings.*

[43] Spector, A. Z., Bloch, J. J., Daniels, D. S., Draves, R. P., Duchamp, D., Eppinger, J. L., Menees, S. G., and Thompson, D. S. *The Camelot Project.* **Database Engineering**, vol. 9 (1986). *Also available as Technical Report CMU-CS-86-166. Carnegie Mellon University, November 1986.*

[44] Staudt, B., Krueger, C., and Garlan, D. *TransformGen : Automating the Maintenance of Structure-Oriented Environments.* Technical Report, no. CMU-CS-88-186, Cambridge, MA, Pittsburgh, PA, November 1988.

[45] Steele, Jr., G. L. **Common Lisp: The Language.** Second Edition. Digital Press, 1990.

[46] Wadlow, M. H. F. H. M. M. T. P. M. *The Alexandria Project: In support of an Information Environment. ITC internal report.*

[47] Wegner, P. and Zdonik, S. B. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* in: **Proccedings of the European Conference on Object-Oriented Programming (ECOOP)**, edited by S. Gjessing and K. Nygaard. **Lecture Notes in Computer Science**, vol. 322, Springer-Verlag, Oslo, Norway, 1988, pp. 55–77. *Also available as BROWN-CS-88-10.*

[48] Zdonik, S. *Object-Oriented Type Evolution.* in: **Advances in Database Programming Languages**, edited by F. Bancilhon and P. Buneman. ACM Press, New York, NY, 1990, pp. 277–288.

[49] Zdonik, S. B. *Version Management in an Object-Oriented Database.* in: **Advanced Programming Environments: Proceedings of an International Workshop**, edited by R. Conradi, T. M. Didriksen, and D. H. Wanvik. **Lecture Notes in Computer Science**, vol. 244, Springer-Verlag, Berlin, 1986, pp. 405–422.

[50] Zicari, R. *A Framework for O_2 Schema Updates.* Rapport Technique, no. 38-89, GIP Altair, Rocquencourt, France, October 1989.

A A Simple Cost Model

This section describes my preliminary attempts at deriving a model to evaluate costs of a particular database consistency strategy. For our current purposes, only the three established schemes, eager and lazy conversion and emulation, were considered. Using this model, more elaborate strategies can be defined and evaluated.

A.1 Data and Access Model

We will consider the database as a heterogeneous collection of tuples, tagged with their type, and addressable by name. Database objects can be accessed in only two ways: directly, by address, or indirectly, via the instance collection associated with each type. Also, the database is not strictly "Object-Oriented", as it lacks both a type hierarchy and procedural information.

A.2 Description of Evolution in a Homogeneous Database

For the time being, let us imagine that the database consists only of instances of a single type. (We will generalize this later.) Let us consider the case of a change to the representation to the instantiated type. In this context, we will evaluate the costs of the three aforementioned strategies.

A.2.1 Eager Conversion

The entire cost of eager conversion of our database is borne at the time the evolution is specified. At this time, the system must convert all the objects in the database. The cost of such a process is:

$$C_{\text{ec}} = d\tau + d\chi$$

where d is the number of objects of the changed type in the database (*i.e.*, the size of the database), τ is the cost required to determine the type of the object, and χ is the cost required to convert an old instance of the type into a new instance.

A.2.2 Emulation

In emulation, all the cost is borne at object-access time. The per-access cost of emulation can be expressed as:

$$C_{\text{em}} = \tau + \epsilon$$

where ϵ is the cost of emulation. (In reality, the cost of emulation would likely depend on the feature of the object being accessed, but this cost is simplified to a single value in this model.)

A.2.3 Lazy Conversion

Under lazy conversion, the cost is deferred at evolution time until the first access of any relevant object. As in the emulation case, the cost can be expressed on a per-access basis. However, unlike emulation, the per-access cost is not fixed, as it depends on the probability of the accessed object having already been accessed since evolution time.

Before we can establish this probablity, we must define a pattern of access on objects in the database. Possibilities for such an access model include:

- A fixed number of objects are repeatedly accessed, while the rest are ignored.
- Objects in the database are accessed uniformly, *i.e.*, on any access, each object stands an equal chance $(1/n)$ of being selected.
- All objects are accessed in turn.

I've selected the uniform access pattern as our model because it is fairly pessimistic, and because, of the three, it lends itself best to extensions later on.

Given a uniform model of object access, we can know derive a cost function for the ath object access since the evolution:

The expected number of distinct objects of the evolved type that have been accessed so far (*i.e.*, since the evolution) can be expressed as the following recursive function:

$$\begin{aligned} e_0 &= 0 \\ e_{a+1} &= e_a + \tfrac{n-e_a}{n} \end{aligned}$$

which can be reduced to:

$$e_a = (1 - (\tfrac{n-1}{n})^a)$$

where a is the total number of database accesses so far.

Given a value for e_a, we can define the cost of access as:

$$\begin{aligned} \mathcal{C}_{lc} &= \tau + (e_{a+1} - e_a)\chi \\ &= \tau + \tfrac{n-e_a}{n}\chi \\ &= \tau + (1 - \tfrac{e_a}{n})\chi \\ &= \tau + (\tfrac{n-1}{n})^a\chi \end{aligned}$$

A.3 Introduction of Multiple Types

We can now generalize our model to support multiple types and multiple type evolutions. We do this by defining the emulation or conversion cost per type; types that have no active evolution have a cost set to 0. The cost functions for eager conversion and emulation now become:

$$\mathcal{C}_{\text{ec}} = N\tau + \sum_{i=1}^{p} n_i \chi_i \quad \text{(total)}$$

$$\mathcal{C}_{\text{em}} = \tau + \frac{1}{N} \sum_{i=1}^{p} n_i \epsilon_i \quad \text{(per-access)}$$

where p is the number of types instantiated in the database, n_i is the number of objects of each type, N is the total number of objects ($N = \sum_{i=1}^{p} n_i$), and χ_i and ϵ_i are the conversion costs, and emulation costs per type.

In generalizing the per-access cost of lazy conversion, our access model selection pays off. Note that since we have employed a uniform access model, the probability that the next object accessed will be of a certain type is $\frac{n_i}{N}$. Given that the type of the object and whether it requires conversion[17] are independent, the (conditional) probability that the object of a certain requires conversion is the simple product of the respective probablities ($\frac{n_i}{N}$ and $(\frac{n_i - 1}{n_i})^{a_i}$). Thus,

$$\mathcal{C}_{\text{lc}} = \tau + \sum_{i=1}^{p} \frac{n_i}{N} \chi_i (\frac{n_i - 1}{n_i})^{a_i} \quad \text{(per-access)}$$

where a_i is the number of accesses of objects of each type.

The model presented here remains very simplistic. A number of important parameters have been omitted, including:

1. The setup cost (consumed at evolution specification time) for the emulation and lazy conversion strategies.

2. Some measure of the *availability expectation factor* of the database, *i.e.*, are the costs associated with having an inaccessible database, as a function of time down.

3. The relationship between the cost of emulation (ϵ_i) and the cost of conversion (χ_i) for any given pair of type-versions.

[17] This is true if we assume all types require conversion, albeit with a possible conversion cost of zero.

B The Rental Problem Made Concrete

One of the canonical problems motivating the study of competitive algorithms is known as the **Tuxedo Rental Problem**[18], which can be described as follows:

> Assume that you will be invited to an unknown number of events in your lifetime that will require you to wear a tuxedo. Let the cost of a tuxedo rental be r per event, and the cost of a (new?) tuxedo be $p > r$. Your goal is to minimize the amount of money you spend on formal attire over the course of your lifetime.

A simple (and nice) solution to the problem, which is guaranteed to be within a factor of 2 of the optimal solution, is this:

> Keep a running total of the amount of money spent to date on tuxedo rental. Rent tuxedoes on a per-event basis until such time as the rental of a tuxedo for the (next) imminent affair would cause you to spend more on rentals than a tuxedo purchase would have cost you had you bought at the beginning.

This form of online algorithm maps quite well into the domain of conversion strategies. Emulation, just like the rental, incurs constant cost per-event, while conversion, like the purchase, incurs a (higher) flat rate and no marginal cost. This has led me to think that online competitive algorithms might assist me in the design of an efficient, pliable, conversion strategy.

The anecdotal strategy described on p. 16 with the three object pools, distinguished by their access frequency, can be implemented using the Rental Problem algorithm. Given the relative costs of emulation and conversion, the implementation is obvious. Here is a simple (albeit space-consuming) algorithm that does not rely on knowledge of costs:

> At type evolution time (*i.e.*, when a new version-type is installed), or when an object of a previous type version is first accessed, perform a conversion and record the amount of (CPU) time it required. For each additional object accessed, maintain a tally of the time expended to emulate (and record!) the operation on it. When the tally exceeds the previously-measured conversion time, convert the object.

[18] **The search for a publication reference for this important problem is in progress.**

For: **Stewart Clamen**

Job Name: **Proposal.english**

Printing Date: Mon Sep 30 03:13:28 1991

Printer: sand / PrintServer 20 (interpreter version 48.3)

School of Computer Science

Reliable Distributed Computing with Avalon/Common Lisp

Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Avalon is a set of linguistic constructs designed to give programmers explicit control over transaction-based processing of atomic objects for fault-tolerant applications. These constructs, designed as extensions to familiar programming languages such as C++ and Common Lisp, are tailored for each base language so the syntax and spirit of each language are maintained. We present here an overview of these novel aspects of Avalon/Common Lisp: (1) support for *remote evaluation* through a new evaluator data type; (2) a generalization of the traditional client/server model of computation, allowing clients to extend server interfaces and server writers to hide aspects of distribution, such as caching, from clients; (3) support for *failure atomicity* through automatic commit and abort processing of transactions; and (4) support for *persistence* through automatic crash recovery of atomic data. These capabilities provide programmers with the flexibility to exploit the semantics of an application to enhance its reliability and efficiency. Avalon/Common Lisp runs on IBM RT's on the Mach operating system. Though the design of Avalon/Common Lisp exploits some of the features of Common Lisp, e.g., its packaging mechanism, all of the constructs are applicable to any Lisp-like language.

1. Introduction

Large networks of computers supporting both local and distributed processing are now commonplace. Application programs running in these environments concurrently access shared, distributed, and possibly replicated data. Examples of such applications include electronic banking, library search and retrieval systems, nation-wide electronic mail systems, and overnight-package delivery systems. Such applications must be designed to cope with failures and concurrency, ensuring that the data they manage remain *consistent*, that is, are neither lost nor corrupted, and *available*, that is, accessible even in the presence of failures such as site crashes and network partitions.

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions must satisfy three properties: serializability, failure atomicity, and persistence. *Serializability* means that transactions appear to execute in some serial order. *Failure atomicity* ("all-or-nothing") means that a transaction either succeeds completely and *commits*, or *aborts* and has no effect. *Persistence* means that the effects of a committed transaction survive failures. We use the term *atomic* to stand for all three properties.

Although transactions are widely used in the database community, demonstrating that they can be a foundation for general purpose distributed systems remains a challenge and is currently of active interest. Appropriate programming language support for application programmers would greatly enhance the usability and thus, generality, of such systems.

Avalon is a set of linguistic constructs designed as extensions to familiar high-level programming languages such as C++ [25] and Common Lisp [16]. The extensions are tailored for each base language, so the syntax and spirit of each language are maintained. The constructs include new encapsulation and abstraction mechanisms, as well as support for concurrency and recovery. The decision to extend existing languages rather than to invent a new language was based on pragmatic considerations. We felt we could focus more effectively on the new and interesting issues such as reliability if we did not have to redesign or reimplement basic language features, and we felt that building on top of widely-used and widely-available languages would facilitate the use of Avalon outside our own research group.

This paper presents an overview of some of the more novel aspects of Avalon/Common Lisp. The distinguishing characteristic of Avalon/Common Lisp, in contrast to Avalon/C++ [6] and other transaction-based distributed programming languages (see Section 6), is its support for *remote evaluation* [23]. Lisp's treatment of code as data provides a natural and easy way to implement remote evaluation since we simply transmit code, as well as data, between clients and servers. Moreover, we exploit remote evaluation to extend and generalize the traditional client/server model of distributed computing. Thus, the programmer gains more flexibility in structuring an application, while often simultaneously improving its performance.

We have implemented the Avalon/Common Lisp constructs presented herein on top of Camelot [21], a distributed transaction management system (written in C) built at Carnegie Mellon. Camelot provides low-level facilities like lock management, two-phase commit protocols, and logging to stable storage.

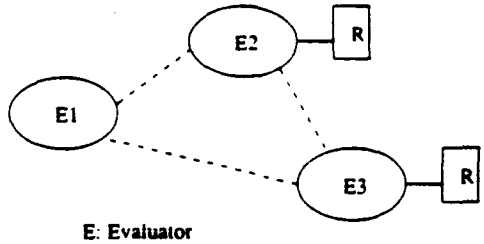
The particular extensions we designed for Common Lisp are applicable to any Lisp-like language, though for concreteness, all our examples will be expressed in Avalon/Common Lisp. We assume the reader has a reading knowledge of Common Lisp.

In Section 2 we give an overview of Avalon/Common Lisp's model of computation and program structure as they relate to distribution, persistence, and concurrency. Sections 3 and 4 explain the novel features of Avalon/Common Lisp related to distribution, in particular remote evaluation and our generalization of the traditional client/server model. Section 5 illustrates features of Avalon/Common Lisp related to persistence. Section 6 compares Avalon/Common Lisp with other transaction-based, distributed programming languages and closes with a summary of our current status.

2. Overview

Distribution

An Avalon/Common Lisp computation executes over a distributed set of *evaluators* (Figure 1), each of which is a distinct Lisp process. An evaluator resides at a single physical site, but each site may be home to multiple evaluators. A user starts a computation at an *initiating* evaluator, which may communicate with other *remote* evaluators. To a first approximation, evaluators communicate through remote procedure calls with call-by-value semantics. The dotted lines in the figure indicate possible call paths between evaluators.



E: Evaluator

R: Recoverable Store

Figure 1: Model of Distributed Evaluators and Recoverable Storage

As in Common Lisp, an Avalon/Common Lisp program consists of a set of *packages*. Each evaluator is host to one or more packages. We map the standard client/server model of distributed computing onto our more general architecture as follows: We put a client's code in one package and execute it on the initiating evaluator, and for each server, we put its code in a separate package and execute it on a remote evaluator.

Section 3 will explain how we extend this standard client/server model by using remote evaluation in combination with the feature that an evaluator can be host to multiple packages. The combination frees us from the above one-to-one correspondences between client code (or server code) and a package, and between a client process (or server process) and an evaluator. In short, in our full extended client/server model, client code can cross evaluator boundaries, can be split into more than one package, or can coexist with server code at the same evaluator. Similar remarks hold for server code.

Persistence

Since Avalon/Common Lisp provides *transactions*, we need to provide a way to support failure atomicity and persistence. When a crash occurs, we need to recover the state of the system to some previously saved consistent state, one that reflects all changes performed by all committed transactions.

Each evaluator has access to at most one private *recoverable store* (see Figure 1), which itself is managed by a separate process.[1] Normally, there would be no recoverable store associated with the evaluator where the client code resides, but there would be one per evaluator that is host to a server.

At the programming language level, each server package encapsulates a set of *object bindings* and exports a set of *functions*. Each object binding is a mapping between a symbol and an object. A binding can be declared to be *persistent*; otherwise it is considered to be *volatile*. Persistent bindings (and the objects to which they map) are allocated from recoverable store; hence, persistent bindings survive crashes, while volatile ones do not. By convention, a server's functions should provide the only means for a client and other servers to gain access to the server's object bindings, and thus its recoverable objects.

It makes sense to access recoverable objects only when executing a transaction so Avalon/Common Lisp provides control primitives to begin, commit and abort transactions. Section 5 shows a use of these primitives.

Concurrency

Avalon/Common Lisp supports concurrent transactions ("heavy-weight" processes), but no concurrency within a transaction. Serializability of transactions is guaranteed by using standard two-phase read/write locks on objects [8]. A transaction holds its locks until it commits or aborts.

Since Common Lisp does not support multiple threads of control, in particular "light-weight" processes as in C Threads [5], we have a simpler model of computation with respect to concurrency than that for other languages such as Avalon/C++. Specifically, only one thread of control executes within an evaluator at once. For example, suppose two clients each make a request at a single server. The (server's)

[1] Each recoverable storage manager is a C process since we currently use Camelot's implementation of recoverable storage; hence, each of our Lisp processes communicates with a C process whenever recoverable storage is accessed.

evaluator processes these two requests serially. On behalf of the first request, it accesses the recoverable store, acquires appropriate read or write locks, and returns appropriate result values. The evaluator then services the second request. If the second request creates a lock conflict, the (server's) evaluator blocks until the lock is freed. Lock conflicts can arise because locks are released as transactions complete, not when function calls return.

Avalon/Common Lisp supports nested transactions, but, again because of the limited kind of concurrency we can support in Common Lisp, each transaction can have at most one active child transaction. A transaction commits only if its child has committed or aborted; a transaction that aborts aborts its child. A transaction's effects become persistent only when it commits at the top level.

The most interesting and novel aspects of Avalon/Common Lisp relate to its way of handling distributed computing, and not persistent storage or concurrency. Thus, the next two sections will focus on the issues related to distribution: remote evaluation and the extended client/server model.

3. Remote Evaluation

3.1. Example Uses of Extensions

Suppose, for simplicity, there are two evaluators, one *local* and one *remote*, where the local evaluator might be the initiating evaluator for some computation. The following expressions:

```
(let ((a 123) (b 45)) (+ a b))
(let ((a 123) (b 45)) (remote (+ a b)))
(let ((a 123) (b 45))
     (remote (+ (local a) (local b))))
```

all return the same value to the user, namely the number `168`. Given that the function `+` refers to the built-in generic addition function, all three expressions have the same semantic meaning. How they differ is where the various subexpressions are evaluated.

In the first expression, all computation (new binding creation, variable lookup, function application) occurs on the local evaluator.

In the second expression, the creation of bindings for `a` and `b` occurs on the local evaluator, while the `remote` special form directs the evaluation of the `(+ a b)` to be performed on the remote evaluator. The lexical environment, containing the local bindings for `a` and `b`, is transmitted along with the expression `(+ a b)` to the remote evaluator.

The evaluation of the third expression occurs similarly to the second, except that the evaluations of the expressions `a` and `b` (within the `(+ (local a) (local b))` expression) are performed back on the local host. Since `+` is already defined on the remote evaluator, this process is equivalent to a traditional remote procedure call (RPC), where the arguments (and not the actual function) are evaluated locally and then transmitted to a remote server for application.

3.2. New Functions, Special Variables, and Macros

As an extension, Avalon/Common Lisp provides one new data type, the **evaluator**, two new special variables, `*remote-evaluator*` and `*local-evaluator*`, and a small number of new special forms, the most important of which are `remote` and `local`. Intuitively, the two forms are used to translate the thread of computation from one evaluator to another, e.g., from the designated local evaluator to some remote evaluator. Below we give the meaning of each in the style of the Common Lisp manual [16].

`make-evaluator` *string* [*Function*]

This function finds and returns the evaluator whose name is specified by the string argument. If none exists, it builds and returns a new evaluator object. Evaluators are *first-class* objects: one can store an evaluator away in other data structures, perform remote evaluations on it at some future time, and transmit them.

`*remote-evaluator*` [*Variable*]

This special variable names the evaluator used to evaluate expressions of the form `(remote` *expr*`)`. On an initiating evaluator, it is bound by default to the initiating evaluator itself until the user changes it to point to some other (remote) evaluator. On a remote evaluator, it is bound by default to the remote evaluator itself. If desired, the programmer can explicitly reset this binding dynamically.

`*local-evaluator*` [*Variable*]

This special variable names the evaluator used to evaluate expressions of the form `(local` *expr*`)`. In the case of an initiating evaluator, it is normally unbound. In the case of a remote evaluator, it is bound by default to the evaluator from which the `remote` was called. If desired, the programmer can explicitly reset this binding dynamically.

`remote` *expr* `&optional` *evaluator* [*Macro*]

This special form's semantics is identical to `identity` except that: (1) The actual computation is performed by the evaluator bound to `*remote-evaluator*` (or to the evaluator specified as the optional argument) with the same lexical environment as the current

evaluator, but a different current package and dynamic state; and (2) the object returned is a *copy* of the result, as opposed to the result object itself. Even in the case where the evaluator bound to `*remote-evaluator*` is specified to be or defaults to the current evaluator, a copy of the resulting object is returned.

Since the process for transmitting data from one evaluator to another necessitates creating copies of objects, mutable objects[2] are not `eq` to their remotely referenced analogues. This is the primary incompatibility introduced by the use of `remote` expressions in a program. Despite the loss of identity, we still preserve sharing of common substructures among transmitted objects, so that values that are comparable on one evaluator are still comparable on another. Hence, we have:

```
(let* ((a (the (not (or number symbol character))
              <arbitrary lisp object>))
       (b a))

  (eq a (remote a))                    ⇒ nil
  (remote (eq a (remote a)))           ⇒ nil

  (equalp a (remote a))                ⇒ t
  (remote (equalp a (remote a)))       ⇒ t

  (remote (eq a b))                    ⇒ t
  (eq (remote a) (remote b))           ⇒ unspecified
```

Here an object that is neither a number, symbol, nor character is locally bound to `a` and `b`. The first two comparisons return `nil` since the object bound to `a` and its copy are different objects, regardless of where the comparison is evaluated. The next two comparisons return `t` because the values of `a`'s object and its copy are the same. The next comparison shows that remotely comparing the identities of `a` and `b` is identical to comparing them locally. Finally, the last comparison shows that while remote and local copies are not identical, the results of different `remote` calls to the same evaluator may return the same object.

`local` *expr* [*Macro*]

This special form has meaning only when evaluated dynamically within a `remote` expression. Its semantics is identical to `identity` except that: (1) Computation occurs at the evaluator specified in `*local-evaluator*`; normally, this is the evaluator where the most dynamically immediate `remote` expression was evaluated; and (2) the object returned is a copy of the object, instead of the object itself.

[2] In Common Lisp, all objects, except for numbers, characters and symbols, are mutable.

Avalon/Common Lisp gives the programmer the flexibility to redirect the thread of computation, if desired, by using the optional parameter to `remote`, or by explicitly setting `*remote-evaluator*` to an evaluator different from the default. Hence, the user can make *third-party calls*, i.e., calls by one remote evaluator to another evaluator. Third-party calls would be common when one server calls another server on behalf of the original computation performed for the client. The calling evaluator is then defined to be the local evaluator and the third evaluator to be the remote evaluator. For example, in Figure 1, if E1 remotely calls E2 which then remotely calls E3, then E3's `*local evaluator*` is automatically set to E2 and its `*remote-evaluator*`, to E3.

Note that since special variables can be set dynamically, they need not reflect the call chain, though normally they would. In the previous scenario, for example, if E3's `*local-evaluator*` is explicitly reset to E1, then `local(...)` expressions would be evaluated at E1, not E2, even though E2 made the remote call to E3. Results are still returned to the evaluator that initiated the `remote` call; hence they would be returned to E2, not E1.

3.3. Abstract Interpreter

Figure 2 shows a simplified abstract interpreter, giving a more formal semantics to the evaluation of the special forms, `remote` and `local`. It does not handle the case of preserving (remote) side effects on shared, mutable objects.

We first define a `dynamic-state` to include the (current) lexical environment, control-related tags and labels, and names of the local and remote evaluators. The lexical environment includes both local variable and local function bindings. We define an `evaluator` to be a name and a set of packages.

To see what `eval` does, we first explain what the helping function `handle-remote` does. It takes four arguments: the expression being evaluated; a dynamic state that includes some lexical environment; and two evaluators, one to indicate where `local` expressions are to be evaluated and one to indicate where `remote` expressions are to be evaluated. A new dynamic state is created and used as the state in which the argument expression is evaluated. The `deep-copy` function preserves internal sharing of objects. It is similar to the `read` of a `print` on printable Common Lisp objects. The recursive calls to `eval` and `deep-copy` ensure that expressions with nested `remote`'s and `local`'s are handled properly.

The `eval` function itself takes three arguments, the expression being evaluated, a dynamic state that includes some lexical environment, and an evaluator. If the expression to be evaluated is a `remote` then first a check is made to see if a specific evaluator is bound to the optional argument in the `remote` call; if not, then the

```
(defstruct dynamic-state
  lexical-env
  catch-tags
  labels
  local-evalr
  remote-evalr
  )

(defstruct evaluator
  name
  packages
  )

(defun eval (expr state evalr)
  (case expr
        ;; other cases ...

        (remote
          (handle-remote (remote-body expr)
                         state evalr
                         (or (remote-evalr expr)
                             (dynamic-state-remote-evalr evalr))))
        (local
          (handle-remote (remote-body expr)
                         state evalr
                         (dynamic-state-local-evalr evalr)))))

(defun handle-remote (expr state oevalr nevalr)
   (deep-copy
     (eval
        expr
        (make-dynamic-state
          :lexical-env  (deep-copy (dynamic-state-lexical-env state))
          :catch-tags   (dynamic-state-catch-tags state)
          :labels       (dynamic-state-labels state)
          :local-evalr  oevalr
          :remote-evalr nevalr
          )
        nevalr)))
```

Figure 2: Abstract Interpreter for Handling Remote Evaluation

remote evaluator bound in the dynamic state is passed as the new remote evaluator to `handle-remote`. Handling a `local` is simpler; the local evaluator bound in the dynamic state is passed as the new remote evaluator to `handle-remote`.

3.4. More Examples

The environment passed as part of a `remote` call does not include the Common Lisp "special" (global) bindings. In the following example:

```
(defvar a 123)
(let ((b 45))
    (remote (+ (local a) b)))
```

an explicit call *back* to the initiating evaluator (using `local`) is required in order to ensure that the special value of a is retrieved; otherwise, the global binding of a on the remote evaluator would be used. Note that the default binding of `*local-evaluator*` will cause `local` to direct a computation back to its originating evaluator.

Since one of Avalon's design goals is to minimize interference with the target language's semantics, nearly all Common Lisp expressions can be "wrapped in" a `remote` to give the desired and expected effects. The lambda expression below is transmitted to the remote evaluator along with its argument for evaluation, illustrating that even procedural objects are permissible within `remote` expressions:

```
(remote ((lambda (x) (* x x)) 4))
```

We also support the remote application of recursively defined functions such as:

```
(labels ((fact (n) (if (< n 2)
                       1 (* n (fact (- n 1))))))
  (remote (fact 20)))
```

since the current lexical environment is transmitted along with the expression. During the evaluation of the above code, the recursive function `fact`, bound in the lexical environment, is applied to 20 on the remote evaluator, and the result is transmitted back to the local evaluator.

The effects of mutating operations in the lexical environment are preserved across evaluator boundaries. For example, the following returns 10:

```
(let ((a 5))
  (remote (setq a 10))
  a)
```

We also handle exits, both local and dynamic, transparently. The result below will be 12, just as if the `remote` call had never existed:

```
(block tag
  (remote (+ 9 (return-from tag 12))))
```

Likewise with the following, the result is also 12:

```
(progn
  (remote (defun add9 (x)
                (+ x (throw 'foo 12))))
  (catch 'foo (remote (add9 1))))
```

3.5. Transmission of Objects

Avalon/Common Lisp supports transmission of all Common Lisp readable types. A type is *readable* if all its instances can be created through the Common Lisp reader using the type's default print representation. Some examples of readable types include `simple-arrays`, `lists`, and `structs`. Most readable types are trivially transmissible since from one evaluator we simply pass an object's print representation and at the other evaluator we reconstitute a copy of the object using the built-in `read` function. We also support transmission of some non-readable Common Lisp types like `functions` and `hash tables`. For a more complex type, like object classes, users would need to define their own *marshall* function, which traverses an object's abstract representation and creates a transmissible version, and *unmarshall* function, which reverses that mapping.

As an optimization, we plan to add to our current implementation support for both partial transmission of large objects and transmission of partially evaluated objects. For large readable objects, such as a complex network of `structs`, we would not copy and transmit the entire object but just its root and its descendants up to *n*-levels deep, where *n* is user-definable. Hence transmitted objects might include *remote references*, i.e., names of objects that reside remotely. Currently our support for partial transmission of large objects is limited to only immutable `structs`. Finally, for conceptually infinite objects akin to *streams* in Scheme [1], we need transmit only a partial evaluation of their values.

4. Extended Client/Server Model

The client/server model is a common paradigm for distributed computing, especially in systems based on remote procedure call. By introducing remote evaluation into Avalon/Common Lisp we can extend this model in useful and powerful ways. In this section, we explore these extensions, by presenting several models of how distributed programs may be structured in Avalon/Common Lisp .

In the traditional client/server model, the RPC interface serves two purposes. It defines both the calling interface between a client and server and the boundary along which a computation is distributed. The caller (client) is also the initiator process of some computation; the callee (server) is also some initiated process executing on behalf of the caller.

Avalon/Common Lisp separates these two functions. We use the terms *client* and *server* to distinguish between the caller and callee. This client/server distinction defines the interface between the facilities provided by the server programmer, and those provided by the client programmer, just as is true for interfaces in non-distributed programs. We use the terms *local* and *remote* to distinguish between the initiator and initiated processes, i.e., evaluators, of a distributed computation. The local/remote distinction serves to define the boundary along which a computation is distributed. A computation is local if it is performed at the evaluator initiating the computation, while it is remote if it occurs at some evaluator different from the initiator. Remote evaluation is the mechanism by which Avalon/Common Lisp expresses this change in computational locus.

In what follows, we suggest alternative ways to organize the remote and local aspects of client and server interfaces, ranging from traditional RPC to a scheme where both the client and the server do computation both remotely and locally.

As a motivating example, we consider a simple distributed database of bibliography entries such as that used for Scribe or LaTex .bib files. We assume that the user of the database is computing on some local site, e.g., a personal workstation, while the database itself resides on

some remote site. The database interface consists of set operations like `intersection` and `union`; a `matches` function that takes as input a query and returns a set of matching bibliography entries; and a `print-bib-entrys` function that takes as input a set of bibliography entries and returns its print representation. Thus, a typical bibliography database user might write:

```
(print-bib-entrys
  (union
    (matches author-named-Edsger)
    (matches author-named-Butler)))
```

to print all the database entries authored by people named Edsger or Butler.

The Traditional Client/Server Model

To get the effects of RPC as used in the traditional client/server model in Avalon/Common Lisp, we simply put a `remote` around the outermost function call. If the database, the set operations, and the printing and matching functions all reside remotely, then the following code fragment shows how our original single-site query would be expressed:

```
(remote
  (print-bib-entrys
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The Extensible Server Model

In Avalon/Common Lisp, a client can extend a server's interface by transmitting function definitions to the server and can then execute them remotely. In our example, the client first uses a remote `defun` to define a more complicated match function:

```
(remote
  (defun match-Edsgers-or-Butlers ()
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The client executes the following code locally, which evaluates the newly defined function remotely:

```
(remote
  (print-bib-entrys (match-Edsgers-or-Butlers)))
```

A client would normally make multiple remote definitions at one time, perhaps as part of its initialization code. There are several advantages to providing extensible servers. The client programmer gains flexibility by tailoring the server interface to the needs of his or her application. Concrete examples of software with extensible interfaces are Emacs and Postscript [26]. The programmer also can greatly enhance the application's performance by allowing a complex computation to take place near the resource it is manipulating. For example, NeWS [13], an extensible windowing system, can support the smooth rubber-banding of spline curves, while X [20], which essentially uses the standard RPC paradigm, has difficulty smoothly rubber-banding even straight lines.

The Hidden Distribution Model

By permitting some or all server code to run locally, that is, at the local evaluator, Avalon/Common Lisp allows clients to be completely unaware of the distributed nature of a computation. Server writers are free to hide some or all of the distributed aspects of the program from a client. In the most extreme case, the client may never even know that it is using a distributed service.

In the hidden distribution model, our example looks as follows. On the local side, the server writer makes the following definitions (we define macros instead of functions to suppress one level of evaluation):

```
(defmacro matches (query)
  `(remote (matches ,query)))

(defmacro union (setA setB)
  `(remote (union ,setA ,setB)))

(defmacro print-bib-entrys (db)
  `(remote (print-bib-entrys ,db)))
```

The client code is the same as for the non-distributed case:

```
(print-bib-entrys
 (union
  (matches author-named-Edsger)
  (matches author-named-Butler)))
```

Here, when the client calls the three functions provided in the local side of the server code, the server makes the explicit remote calls to the remote side of the server code.

The Full Model

The full model allows both the client and the server to compute both at the local and the remote evaluators. Figure 3 depicts this situation where again, the dotted lines indicate possible call paths. Support for this generality is useful if we want both the ability t

perform complex client computations at the remote site, and to allow the server to hide key aspects of the distributed computation, such as caching.

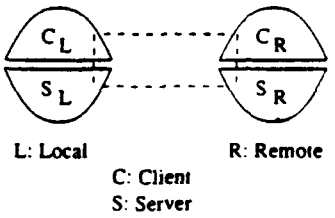


Figure 3: The Full Extended Client/Server Model of Distributed Evaluators

To illustrate both of these capabilities, suppose the server writer implements the `matches` and `union` functions to manipulate the database entries using some compact, but incomplete representation of each entry, while `print-bib-entrys` must have the entire entry before printing it. Caching the complete entries at the local site prevents them from being repeatedly shipped from evaluator to evaluator (i.e., remote to local) while hiding this caching in the server interface allows the client to ignore the complications introduced by the cache.

For our example, on the local side, the server writer makes the following definitions:

```
(defun print-bib-entrys (s)
  (set-map #'print-db-entry s))
(defun print-db-entry (set-entry)
  (unless (value-cached-p set-entry)
    (add-to-cache (get-remote-object set-entry)))
  (print-entry (cached-value set-entry)))
```

The client makes the following remote definition (as before in the Extensible Server Model):

```
(remote
  (defun match-Edsgers-or-Butlers ()
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The client executes it:

```
(print-bib-entrys (remote
            (match-Edsgers-or-Butlers)))
```

The reader should compare the above call to that used in the Extensible Server Model.

The above example illustrates what would commonly be done for querying a large database. In general, application programmers need the ability to write *split queries*, where part of the query is performed remotely through a server interface, and part performed locally through client code. A typical query might be split into a *search predicate* executed remotely and a *filter predicate* executed locally. For example, the search predicate might return a stream of bibliography entries to the client who might then further filter out every fifth entry.

5. Persistence

In this section we show how Avalon/Common Lisp supports failure atomicity through the `with-transaction` construct and persistence through declarations of persistent bindings. We first illustrate these features by showing the relevant pieces of the package for the bibliography database server.

5.1. Example Uses of Extensions

Here we make the database's binding persistent and initialize it:

```
(defpersistent $bib-database$
               (make-persistent (empty-set)))
```

By convention, we use the "$" characters to distinguish those symbols used for persistent bindings from those used for volatile ones. `Make-persistent` creates a recoverable object; `defpersistent` defines `$bib-database$` as a special symbol whose binding is recoverable, and creates a binding between `$bib-database$` and the recoverable empty set.

We use transactions for standard database operations such as adding, modifying, and deleting entries. Consider the function for adding a bibliography entry:

```
(defun add-bib-entry (entry)
  (with-transaction
   (if (valid-bib-entry-p entry)
       (adjoin $bib-database$
               (make-persistent entry))
       (abort-transaction 'invalid-bib-entry))))
```

If the entry is valid, i.e., well-formed and not already in the database, then we make the volatile value of the `entry` argument persistent and add it to the database. Otherwise, we abort the transaction signalling

the abort condition `invalid-bib-entry`. Since the update is done within a transaction, if a crash occurs during the update, the state of the bibliography database will be as if the update never occurred; Camelot's recovery algorithm will guarantee the database is restored to a previously saved consistent state.

The counterpart to `make-persistent` is `make-volatile`. Since an evaluator communicates with a recoverable store, retrieving a persistent binding from it gives us a handle on a recoverable object. Upon retrieval, we are free to continue to use the object as a recoverable object until we need to either call a standard Common Lisp function or transmit the object back to the local evaluator. Thus as a server writer, we have some latitude as to when to make the `make-volatile` call. For example, both `print-bib-entrys` below have the same eventual effect:

```
(defun print-bib-entrys ()
  (set-mapc #'(lambda (set-entry)
                (print-bib-entry
                  (make-volatile set-entry)))
                  $bib-database$))

(defun print-bib-entrys ()
  (set-mapc #'print-bib-entry
                (make-volatile $bib-database$)))
```

In the first version, `set-mapc` operates on a persistent set (and uses `rec-car`, `rec-cdr`, etc. to traverse the `$bib-database$`[3]). In the second, `set-mapc` operates on a volatile object. `Make-persistent` and `make-volatile` are each idempotent and are inverses of each other.

Avalon/Common Lisp currently supports recoverable versions of a large subset of Common Lisp's built-in types, e.g., `fixnum`, `list`, `simple-string`, `simple-vector`, as well as any type constructed using `struct`'s.

5.2. New Macros and Functions

Here is the programmer's interface to the new macros and functions:

`defpersistent` *variable* [*initial-value*] [*Macro*]

This form is similar to the `defvar` form, except that any binding to *variable* is recoverable, i.e., survives crashes and supports failure atomicity. If given, *initial-value* is assigned to *variable*, as long as *variable* has not previously been bound. *Initial-value* must evaluate to a recoverable object and is only evaluated if it is used to initialize the binding.

All subsequent `setq` operations to *variable* will change the binding atomically; `setq` operations to persistent variables can be aborted if evaluated within a transaction.

`make-persistent` *object* [*Function*]
`make-volatile` *object* [*Function*]

These functions create a persistent (volatile) representation of *object*. If *object* is already persistent (volatile), it is returned as the result.

`with-transaction` *body* [*Macro*]
`with-top-level-transaction` *body* [*Macro*]

Both forms initiate a new transaction and evaluate *body*. `With-transaction`, if evaluated dynamically within another transaction, will begin a nested transaction; otherwise it starts a top-level transaction. `With-top-level-transaction` always initiates a new top-level transaction. Both forms return a mutiple value consisting of a status signifying whether or not the transaction committed, and the result of the last expression in *body*.

Normal evaluation of either form results in a committed transaction. Exceptional exits from the *body* (via `catch/throw` and local exits) result in the transaction aborting. Transactions can also be explicitly aborted via use of `abort-transaction`.

`abort-transaction` *retval* `&optional top-level`[*Macro*]

This form aborts the currently executing transaction. If the optional argument is nil (the default), the innermost dynamically nested transaction is aborted and the value of *retval* is returned as the status in the multiple-value result of `with-transaction`. Otherwise, the current (dynamically scoped) top-level transaction is aborted.

6. Related Work and Discussion

Our work on remote evaluation is closest in spirit to Stamos's Ph.D. work [23] for which he designed extensions to the programming language CLU to support remote evaluation in the context of atomic transactions. Since the target languages differ, so do our concrete designs. We designed and packaged our language extensions in a way that avoids modifying the compiler and instead exploits the interpretative programming style of Common Lisp. Since CLU is a compile-time (strongly) typed language, Stamos defines static checks

[3] Avalon/Common Lisp supports "recoverable" versions of some standard Lisp functions like `car`, `cdr`, `eq`, `eql`, etc. They operate on objects retrieved from recoverable store, rather than normal non-recoverable Lisp objects.

that must be performed to ensure a remote evaluation request is valid. Client extensions to servers and code arguments further complicate both these checks and the compiler's subsequent encoding of a remote evaluation request. We avoid some of these difficulties since our new **evaluator** data type gives us not only a run-time boundary (each is a process), but a compile-time boundary (each defines a global namespace for a set of packages).

Our extensions to the client/server model are similar to that supported by Falcone's Heterogeneous Distributed System architecture, prototyped at DEC [10]. Falcone focuses on support at the operating-system level, rather than at the programming-language level, though he does provide a small Lisp-like language interface to the system facilities. By our extending Common Lisp rather than defining a new language, we have the advantage of completely integrating our extensions with an existing, familiar, and widely-available programming language. Also, Falcone handles only primitive data types such as lists and byte vectors, and does not address persistent and recoverable storage of data.

Avalon/Common Lisp is distinct from other distributed programming languages such as CSP [15], SR [2], Linda [12], Nil [24], and Ada [19], since we have direct support not only for remote evaluation, but for transactions, and in particular the following features: commit and abort processing, crash recovery, atomic objects, and management of persistent data.

Even though Avalon/Common Lisp lacks light-weight processes, there are some similarities between it and other "concurrent Lisp" efforts such as Qlisp [11] and Multilisp [11], both of which support concurrent computation using light-weight, shared-memory processes. Qlisp's *qlambda* construct creates a closure and a process that will be used to evaluate any future application of the closure. Like qlambda instances, Avalon/Common Lisp's evaluators are each identified with a separate process. The lexical environment inherited in a `remote` call could be passed to a Qlisp process as an argument, and the evaluator-specific global environment could be simulated using the environment of the qlambda's closure. Avalon/Common Lisp's `remote` construct is also similar to Multilisp's *future* construct. Both allow the programmer to dispatch arbitrary forms to another process for evaluation. A key difference is that Multilisp's processes, being light-weight, are created on-demand, while the evaluators in Avalon/Common Lisp are heavy-weight, and, therefore, long-lived.

Transactions themselves have been a primary focus in both distributed and centralized data bases ([3], [8], [14], [9]). Several research projects have chosen transactions as the foundation for constructing reliable general-purpose distributed programs, including Argus [17], Arjuna [7], Clouds [18], TABS [22], and Camelot [21]. Of these projects, however, only Argus and Arjuna have addressed the linguistic aspects of the problem. Argus extends CLU and Arjuna extends C++. None of these projects have direct support for remote evaluation or our extended client/server model.

Avalon/C++ and Avalon/Common Lisp differ in significant ways even though they address the same application domain, reliable distributed computing, and are motivated by the need to provide programming level support for transactions. Avalon/C++'s primary design focus was on user-defined atomic data types, in particular, support for *hybrid atomicity*. Programmers can define (hybrid atomic) objects that provide higher degrees of concurrency than that provided by using standard two-phase read/write locks, such as that used for Avalon/Common Lisp. In contrast, Avalon/Common Lisp's primary design focus is on remote evaluation and support for a client/server model more general and flexible than the traditional one such as that used for Avalon/C++. Thus, Avalon/Common Lisp relies on well-known techniques for dealing with serializability (read/write locks) and persistence (write-ahead logging, recoverable virtual memory), but introduces a new model for distributed computing.

Currently, all Avalon software runs on IBM RT's in the Mach and Camelot environments. Avalon/C++ runs on Sun's and Vaxes as well. Avalon/C++ has been operational for a year and we are not doing any further design or implementation work with it.

Avalon/Common Lisp is nearly complete as of this writing. All Avalon/Common Lisp code presented in this paper runs. Besides the bibliography database, other Avalon/Common Lisp examples include a simple array server and a factory-parts database. Further details on the Avalon/Common Lisp programmer's interface are in [4].

In summary, Avalon is a set of linguistic constructs that extend the capability of existing programming languages by directly supporting transactions. For each of our target languages, C++ and Common Lisp, we designed our extensions to be unintrusive and modular. For example, a Common Lisp programmer can load one set of packages if support for only remote evaluation is desired, a different set if support for only recoverable store is desired, or both sets if both features are desired. These language extensions relieve users from the burden of doing low-level system activities such as locking and managing stable storage, and instead allow them to concentrate on the logic required of their application. At the same time, however, they are given enough flexibility to exploit the semantics of their applications to increase their programs' reliability and efficiency.

Acknowledgments

We thank Gene Rollins who has helped motivate some of the design of Avalon/Common Lisp through his willingness to be our first user. He and Karen Kietzke also gave useful comments on earlier drafts of this paper.

We also thank Dave McDonald for providing us with a barebones Common Lisp interface to Camelot's recoverable storage manager, and Alfred Spector and the rest of the former Camelot crew for providing us with basic run-time support.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract No. F33615-87-C-1499. Additional support was provided in part for J.M. Wing by the National Science Foundation under grant CCR-8620027 and for S.M. Clamen by the Office for Naval Research under grant ONR-N00014-88-12-0641.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, the Office for Naval Research or the U.S. Government.

References

[1] H. Abelson and G.J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] G.R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[3] P.A. Bernstein and N. Goodman. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[4] S.M. Clamen, L.D. Leibengood, S.M. Nettles, and J.M. Wing. The avalon/common lisp programmer's guide. 1989. Avalon Design Note 14.

[5] Eric C. Cooper and Richard P. Draves. *C Threads*. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[6] D.L. Detlefs, M.P. Herlihy, and J.M. Wing. Inheritance of synchronization and recovery properties in avalonc++. *IEEE Computer*, 57–69, December 1988.

[7] G. Dixon and S.K. Shrivastava. Exploiting type inheritance facilities to implement recoverability in object based systems. In *Proceedings of the Sixth IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1987.

[8] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in a database system. *Communications ACM*, 19(11):624–633, November 1976.

[9] B.G. Lindsay et al. *Notes on Distributed Databases*. Technical Report RJ2571, IBM San Jose Research Laboratory, July 1979.

[10] J.R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5(4), November 1987.

[11] R. P. Gabriel and J. McCarthy. Queue-based multi-processing lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[12] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[13] J. Gosling, D.S.H. Rosenthal, and M. J. Arden. *The NeWS Book: An Introduction to the Networked Extensible Window System*. Springer-Verlag, 1989.

[14] J.N. Gray. *Notes on Database Operating Systems*, pages 393–481. Springer-Verlag, Berlin, 1978.

[15] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[16] G. Steele Jr. *Common LISP*. Digital Press, 1984.

[17] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[18] M.S. McKendry. Clouds: a fault-tolerant disributed operating system. *IEEE Tech. Com. Distributed Processing Newsletter*, 2(6), June 1984.

[19] Dept. of Defense. Reference manual for the ada programming language. 1983. ANSI/MIL-STD-1815A-1983.

[20] R.W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[21] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, and D.S. Thompson. The camelot project. *Database Engineering*, 9(4), December 1986. Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[22] A.Z. Spector, J. Butcher, D.S. Daniels, J.L. Eppinger D.J. Duchamp, C.E. Fineman, A. Heddaya, and P.M. Schwarz. Support for distributed transactions in the tabs prototype. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[23] J.W. Stamos. *Remote Evaluation*. Technical Report 354, MIT Labxooratory for Computer Science, January 1986.

[24] R.E. Strom and S. Yemini. Nil: an integrated language and system for distributed programming. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983. Also an IBM research report (RC 9499 (44100)) from April 1983.

[25] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[26] Adobe Systems. *Postscript Language Reference Manual*. Addison-Wesley, 1985.

Carnegie Mellon University
School of Computer Science

Avalon Note 16 January 22, 1990

Assessment of the Avalon/Common Lisp Implementation

Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing

1. Introduction

This note assesses the Avalon/Common Lisp implementation effort. Shortcomings in CMU Common Lisp, Camelot's Lisp interface and our own design decisions resulted in an unnecessarily fragile and complex system. We will examine each area in turn, as well as note where the CMU Common Lisp – Camelot platform made implementation easy.

2. CMU Common Lisp

CMU Common Lisp provided a number of features which permitted rapid development of key pieces of the system. Some of these attributes were specific to the Common Lisp language, others were specific to the CMU implementation [4]. The language features which proved to be most useful are listed below.

- Lisp's syntax is easily extended. Augmenting the existing language was trivial.
- Lisp's treatment of code as data made remote evaluation easy to implement since both code and data were readily transmitted between evaluators.
- The Lisp reader and printer provided much of the marshalling functionality for free.
- The ability to incrementally add definitions to a package made it easy for clients to extend a server's interface.
- First class conditions and dynamically bound condition handlers made it trivial to implement Avalon/Common Lisp's exception semantics and to properly handle non-local transfers of control.

Several CMU-specific features were heavily exploited. These included:

- The interprocess communication support provided by Matchmaker and the CMU Common Lisp scheduler was used extensively. Matchmaker generated interfaces from specifications while the scheduler directed the receipt and processing of IPC messages. Hemlock's [3] use of these facilities to support limited remote evaluation and compilation provided a starting point for our development efforts.

- Common Lisp source code and local expertise were readily available. This made it both possible and reasonably easy to extend the semantics of Common Lisp's primitives (ex. `defstruct`).

While CMU Common Lisp provided an attractive environment for prototyping, it did not provide a suitable environment for long term development. The language deficiencies included:

- **Global package namespace.** Each Lisp process maintains a single package namespace, with no security features. Without careful design, clients of a remote evaluator can inadvertently modify each other's definitions.
- **Symbol semantics.** Lisp's read-time symbol lookup and symbol creation, inclusion of a package name as part of a symbol's representation, and a design decision to transmit both a symbol's name and package combined to require a server's exported symbols to be defined on both the initiating and remote evaluators [1].
- **Language size.** Lisp's size and lack of formal semantics make it difficult to rigorously evaluate and specify proposed language extensions.

While the language deficiencies proved to be annoying, the implementation deficiencies were more limiting. These included:

- **No concurrency.** CMU Common Lisp supports only a single thread of control. In particular, "lightweight" process creation, protection of critical regions with mutexes and synchronization using condition variables is not supported. These limitations posed problems for both the remote evaluation and Camelot Lisp implementations.

  In the case of remote evaluation, the lack of support for multiple threads of control combined with an inability to save the current state of a computation restricted evaluators to processing requests serially. Once processing begins on a request, it is taken to completion. An initiating evaluator blocks until it receives a reply. Hence, there is no concurrency at the level of a single evaluator.

  Third party calls and transactions exacerbate the problem. When third party calls are made, all intermediate evaluators in the call chain block. At this point, even the coarse-grained concurrency of different evaluators concurrently executing requests is limited. Client initiated transactions also add to the problem. A transaction may include several `remote` expressions and thus may outlive a single server call. Because locks are released as transactions complete, not when a remote call returns, the probabilities of lock conflicts and deadlock escalate.

  While implementing some priority based request handling scheme and restricting transactions to a single server would reduce the deadlock problems, these are stopgap measures at best. Eliminating useful functionality to regain a limited degree of concurrency is not an acceptable solution.

- **No continuations.** The absence of first-class continuations assumes added importance in light of the lack of concurrency. Continuations provide a mechanism for implementing coroutines, thereby allowing us to effect concurrency without multithreading. In addition, storable continuations are one of the elements needed to provide checkpointable workspaces. The lack of continuations eliminates interesting avenues of exploration.

3. Camelot

Camelot's distributed transaction support and Lisp interface to recoverable storage provided us with considerable functionality. Little additional investment was required to explore transactions and persistence in a Lisp environment. Two Lisp packages, `camelot` and `lro`, and a C executable, the lisp recoverable object (lrec) server, form the core of Camelot's Lisp interface. Before discussing our experiences with this interface, it may be helpful to review each component's role.

`Camelot` exports a set of macros that permit Lisp processes to function as Camelot clients. `Do-transaction` and `abort-transaction` bundle the necessary calls to the Camelot transaction manager to begin, end and kill transactions. `Server-call` invokes operations exported by Camelot data servers.

Note that while this interface provides Lisp processes with access to Camelot, the processes may only participate as clients, not servers. No Lisp equivalent to the Camelot C library exists. McDonald [2] cites several difficulties in developing such a library: servers require multiple threads of control and primitives that guarantee exclusive access to shared data; and CMU Common Lisp uses most of the address space of the IBM RT, thereby limiting the amount of recoverable storage that can be conveniently mapped into the Lisp address space.

Given only the `camelot` package, Lisp's access to recoverable storage is restricted to operations exported by C data servers. The remaining components of the Lisp interface, `lro` and the lrec server, attempt to address this limitation.

The lrec server is a full fledged data server, written using the Camelot C library. It operates on chunks of recoverable, untyped bytes which are referenced by unique object identifiers. `Lrec_malloc` allocates a specified number of bytes and returns a unique object id; `lrec_rec` and `lrec_modify` read and modify a specified number of bytes after acquiring appropriate locks; `lrec_checkin` associates a name with an object id; and `lrec_lookup` maps names to object identifiers.

`Lro` exports an interface which permits Lisp users to manipulate recoverable objects without directly accessing the lrec server. It is at the `lro` level that types are associated with recoverable objects. For example, `lro:rec-malloc` takes a type and name, allocates storage in the lrec server, associates name with the lrec server object, and returns a CLOS instance which provides a handle to the object. This CLOS instance contains all the information needed to retrieve the object from the lrec server and coerce the returned bits to the correct Lisp type. Existing recoverable objects may be accessed by providing `lro:rec-connect` with the object's name and type. A handle is returned.

The problems we encountered in using this interface are itemized below. Instances where our design decisions created or contributed to the problem are noted.

- **Manual recovery.** Camelot's Lisp interface provided no support for automated recovery. Applications must explicitly recover any objects of interest by providing `lro:rec-connect` with an object's name and type. Avalon/Common Lisp automated the recovery process by adding support for declaring and maintaining persistent bindings. Among other things, this required type information and a symbol to object identifier mapping to be maintained in recoverable storage. Support for automatic recovery was implemented entirely at the Lisp level rather than redesigning the lrec server to operate on typed objects. This decision

noticeably impacted performance since multiple transactions and Irec server calls were now required to create a persistent object.

- **Disjoint recoverable and volatile heaps.** Recoverable storage is managed by a C process which requires space to be explicitly malloced and freed. Volatile storage is dynamically allocated and garbage collected by a Lisp process. This storage division is painfully obvious to the user. Operations on persistent objects are perceptibly slower. Also, standard Lisp functions cannot be directly applied to recoverable objects; a persistent object must be coerced to a volatile form or a "recoverable" version of the desired function must be applied.

- **Mismatched clients and servers.** From Camelot's perspective, only the Irec servers are legitimate data servers; all Lisp processes are clients. From Avalon/Common Lisp's perspective, a server is a Lisp process plus its associated Irec server. These differing views, combined with a design decision to support distributed transactions (from ACL clients to ACL servers), resulted in our using Camelot in a way its designers never intended. A transaction containing a `remote` typically has three participants, two Camelot clients (the initiating and remote evaluators) and one Camelot server (Irec). This notion of a single transaction having multiple clients is foreign to Camelot. One consequence of this is that only one client gets notified of aborts, when several clients may need the information. To insure that all interested parties are notified of transaction aborts, initiating evaluators maintain a list of all evaluators visited under the scope of a particular transaction. Each evaluator on the list is notified upon transaction abort.

4. Conclusions

In retrospect, better design decisions could have alleviated some implementation problems. In particular, limiting transactions to a single evaluator reduces the potential for deadlocks and eliminates the Avalon/Common Lisp-to-Camelot server mismatch. Addressing some of the recoverable storage deficiencies at the Irec server level rather than at the Lisp level would provide improved performance and simpler code.

CMU Common Lisp's lack of concurrency hobbled the implementation in a multitude of ways. In addition, Lisp provided a poor language foundation to build on. It was much too easy to become sloppy in our thinking.

If Camelot is to be used as a platform, a Lisp equivalent to the Camelot C library must be built. Substantial additional effort is required to get the object-oriented, garbage collected view of recoverable storage we ultimately want.

CMU Common Lisp and Camelot both provided functionality which enabled rapid development of a prototype system. However, neither provide an appropriate foundation for future work.

References

[1] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, Jeannette M. Wing, *A Programmer's Guide to Avalon/Common Lisp*, Avalon Note 15, January 1990.

[2] Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector, editors, *Guide to the Camelot Distributed Transaction Facility including the Avalon Language*, Draft of September 18, 1989.

[3] Robert A. MacLachlan, Bill Chiles, *Hemlock User's Manual*, Technical Report CMU-CS-89-133, April, 1989.

[4] David B. McDonald, (ed.) *CMU Common Lisp User's Manual - Mach/IBM RT PC Edition*, Technical Report CMU-CS-89-132, April, 1989.

[5] Guy L. Steele Jr., *Common LISP: The Language*, Digital Press, 1984.

Carnegie Mellon University
School of Computer Science

Avalon Note 15 January 22, 1990

A Programmer's Guide to Avalon/Common Lisp

Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing

1. Introduction

Avalon/Common Lisp provides a set of language extensions enabling programmers to develop distributed, fault-tolerant applications. It supports remote evaluation, failure atomicity, and persistence through a collection of packages which export a small number of forms. These packages can be mixed and matched to obtain the desired combination of functionality. This note describes the interface exported by each Avalon/Common Lisp package.

We begin by taking a brief look at Avalon/Common Lisp's architecture. Our intent is to provide just enough background so that terms and concepts used later in the interface descriptions make sense. Clamen et al., [1] provides a more complete description of our computational model and design. Readers familiar with our model can skip immediately to Section 3.

Section 3 introduces an application which we use as a source of examples throughout the note. Sections 4.1 and 6.1 present sample uses of three key packages, `remote-eval`, `rstore` and `trans`, providing support for distributed computation, persistence and failure atomicity, respectively. Sections 4.2, 6.2 and 7.1 define their exported interfaces. We also supply two other packages: The `transmit` package exports forms that permit user-defined transmission functions and the `avalon` package provides a convenient way of accessing Avalon/Common Lisp's complete functionality. Sections 5.1 and 5.2 demonstrate uses of the basic and extended transmission facilities; section 5.3 defines the `transmit` package interface.

The note concludes with a summary of the implementation's status and instructions for running Avalon/Common Lisp.

2. Overview

Distribution

An Avalon/Common Lisp computation executes over a distributed set of *evaluators* (Figure 1), each of which is a distinct Lisp process. An evaluator resides at a single physical site, but each site may be home to multiple evaluators. A user starts a computation at an *initiating* evaluator, which may communicate with other *remote* evaluators. To a first approximation, evaluators communicate through remote procedure calls with call-by-value semantics. The dotted lines in the figure indicate possible call paths between evaluators.

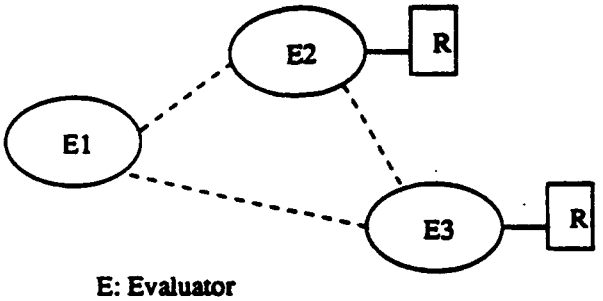


Figure 1: Model of Distributed Evaluators and Recoverable Storage

As in Common Lisp, an Avalon/Common Lisp program consists of a set of *packages*. Each evaluator is host to one or more packages. The standard client/server model of distributed computing maps onto our more general architecture as follows: We put a client's code in one package and execute it on the initiating evaluator, and for each server, we put its code in a separate package and execute it on a remote evaluator.

This standard client/server model can be extended by using remote evaluation in combination with the feature that an evaluator can be host to multiple packages [1]. The examples in this note will focus on the standard model.

Persistence

Since Avalon/Common Lisp provides *transactions*, we need to provide a way to support failure atomicity and persistence. When a crash occurs, we need to recover the state of the system to some previously saved consistent state, one that reflects all changes performed by all committed transactions.

Each evaluator has access to at most one private *recoverable store* (see Figure 1), which itself is managed by a separate process.[1] Normally, there would be no recoverable store associated with the evaluator where the client code resides, but there would be one per evaluator that is host to a server.

At the programming language level, each server package encapsulates a set of *object bindings* and exports a set of *forms*. Each object binding is a mapping between a symbol and an object. A binding can be declared to be *persistent*; otherwise it is considered to be *volatile*. Persistent bindings (and the objects to which they map) are allocated from recoverable store; hence, persistent bindings survive crashes, while volatile ones do not. By convention, a server's functions should provide the only means for a client and other servers to gain access to the server's object bindings, and thus its recoverable objects.

It makes sense to access recoverable objects only when executing a transaction, so Avalon/Common Lisp

[1] Each recoverable storage manager is a C process since we currently use Camelot's implementation of recoverable storage; hence, each of our Lisp processes communicates with a C process whenever recoverable storage is accessed.

provides control primitives to begin, commit and abort transactions.

Concurrency

Avalon/Common Lisp supports concurrent transactions ("heavy-weight" processes), but no concurrency within a transaction. Serializability of transactions is guaranteed by using standard two-phase read/write locks on objects [5]. A transaction holds its locks until it commits or aborts.

Common Lisp's lack of support for multiple threads of control, in particular "light-weight" processes as in C Threads [3], simplifies our model of concurrency. Specifically, only one thread of control executes within an evaluator at once. For example, suppose two clients each make a request at a single server. The server's evaluator processes these two requests serially. On behalf of the first request, it accesses the recoverable store, acquires appropriate read or write locks, and returns appropriate result values. The evaluator then services the second request. If the second request creates a lock conflict, the server's evaluator blocks until the lock is freed. Lock conflicts can arise because locks are released as transactions complete, not when function calls return.

Avalon/Common Lisp supports nested transactions, but each transaction can have at most one active child transaction. A transaction commits only if its child has committed or aborted; a transaction that aborts, aborts its child. A transaction's effects become persistent only when it commits at the top level.

3. Sample Application

We will draw examples from a single application, that of a simple distributed database of bibliography entries.

Assume the database user is computing on some local site, e.g., at a personal workstation, while the database itself resides on some remote site. The database interface consists of set operations like **intersection** and **union**; a **matches** function that takes as input a query and returns a set of matching bibliography entries; and a **print-bib-entrys** function that takes as input a set of bibliography entries and returns its print representation. In the single site case, a bibliography database user might write:

```
(print-bib-entrys
  (union
    (matches author-named-Edsger)
    (matches author-named-Butler)))
```

to print all the database entries authored by people named Edsger or Butler.

4. Remote Evaluation

We begin our discussion of the **remote-eval** package by showing how its exported forms might be used to provide distributed access to our bibliography database.

4.1. Example Uses of Extensions

The server's evaluator loads the database, set operations, and printing and matching functions. As the final step in its initialization process, it registers its name with the system name server:

```
(init-evaluator "bib")
```

When this call completes, the server is available to accept requests. A client wishing to make a query executes

```
(setq *remote-evaluator* (find-evaluator "bib"))
```

to find an evaluator which has registered as a bibliography server and stores the returned handle for use in subsequent requests. To make a query, the client simply wraps the outermost function call with a **remote**.

```
(remote
  (print-bib-entrys
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

By default, the evaluator bound to the special variable ***remote-evaluator*** performs the query, using its current package and dynamic state and the client's lexical environment. A *copy* of the result is returned to the client. Supplying a second argument to **remote** explicitly directs computation to the specified evaluator.

It should be noted that output sent to ***standard-output*** as part of **remote** processing, prints on the remote, not the initiating evaluator. In cases where the client needs this generated output, steps must be taken to capture and return the items of interest. As an example, consider the definition for the **print-bib-entrys** function used above.

```
(defun print-bib-entrys (set)
  (with-output-to-string (s)
    (let ((*standard-output* s))
      (mapc #'print-bib-entry set))))
```

The helper function **print-bib-entry** takes a single entry as input and outputs a formatted version of the entry to ***standard-output***. **Print-bib-entrys** temporarily rebinds ***standard-output*** to a stream that saves its output in a string, successively applies the helper function to each element of the set and returns the string containing the saved output as its result.

With the exception of the copy and input/output semantics mentioned above, nearly all Common Lisp expressions can be wrapped in a **remote** to give the expected effects. In particular, both

local and dynamic exits are handled transparently. In the example below, a client searches for all bibliography entries having the keyword "object-oriented". Rather than transmit large amounts of data on searches that were not sufficiently constrained, an appropriate message is thrown back to the initiating evaluator. Hence, the value returned by the catch is either a listing of matching bibliography entries or a string indicating the number of matches found.

```
(catch 'too-many-matches
  (remote
    (let ((set (matches has-key-object-oriented)))
      (when (> (size set) 500)
          (throw 'too-many-matches
                 (format nil "Found ~S matches. Narrow query." (size set))))
      (print-bib-entrys set))))
```

Similarly, all errors and signaled conditions resulting from evaluation on a remote evaluator are passed back to the initiating evaluator, where they are re-signaled. The following shows how to handle the case of too many matches.

```
(remote
  (let ((set (matches has-key-object-oriented)))
    (when (> (size set) 500)
      (error "Found ~S matches. Narrow query." (size set)))
    (print-bib-entrys set)))
```

If a condition contains non-transmissible data, an error message is generated and transmitted back as a string.

4.2. Exported Forms

The **remote-eval** package defines one new data type, *evaluator handle*, and exports the following forms, presented in the style of the Common Lisp manual [6].

remote-eval:init-evaluator *name* [*Function*]

This function initializes and registers the calling evaluator with the network name server. *Name* is a string that will be used by other evaluators to locate the caller. An error is signaled if the registration fails or if the evaluator is already registered under a different name.

remote-eval:find-evaluator *name* **&optional** *host* [*Function*]

This function finds an existing evaluator, registered under *name*, that resides on *host*. Both *name* and *host* are strings, where *host* must be a valid host name. By default, *host* refers to the caller's machine. An evaluator handle is returned. If no matching evaluator is found, an error is signaled.

remote-eval:*remote-evaluator* [*Variable*]

This special variable names the evaluator used to evaluate expressions of the form (**remote** *expr*). On an initiating evaluator, it is bound by default to the initiating evaluator itself until the user changes it to point to some other (remote) evaluator. On a remote evaluator, it is bound by default to the remote evaluator itself.

remote-eval:remote *expr* **&optional** *evaluator* [*Macro*]

This special form does most of the work. Semantically, it is identical to **identity** except that: (1) The actual computation is performed by the evaluator bound to ***remote-evaluator*** (or to the evaluator specified as the optional argument) with the same lexical environment as the current evaluator, but a different current package and dynamic state; and (2) the object returned is a *copy* of the result, as opposed to the result object itself. Even in the case where the evaluator bound to ***remote-evaluator*** is specified to be or defaults to the current evaluator, a copy of the resulting object is returned.

It is an error for *expr* to contain reads to ***terminal-io***.[2] In the absence of callback support, such reads cause deadlock. **Write**s or **print**s to ***terminal-io*** send output to the remote evaluator. If this output is of interest to the initiating evaluator, either the server or client must explicitly capture and return it.

Finally, the **remote** facility ignores non-transmissible objects in the initiating evaluator's lexical environment. If the remote evaluator references a non-transmissible component, an error is signaled.

5. Transmission of Objects

The **transmit** package provides basic transmission support for the **remote** facility and exports forms that permit user-defined transmission functions. A user may provide a *marshall* function, which traverses an object's abstract representation and creates a transmissible version, and an *unmarshall* function, which reverses that mapping.

We begin by discussing Avalon/Common Lisp's default transmission support. **Transmit**'s handling of both symbols and structures impacts the construction of **remote** expressions in non-intuitive ways. Section 5.1 outlines the issues and presents successful and unsuccessful approaches to addressing the problem.

We follow with a look at sample uses of **transmit**'s exported forms, and conclude our discussion by defining each of the package's interfaces.

5.1. Example Uses of Basic Transmission Support

Transmit provides transmission support for all readable plus a few non-readable types (see Table 1). A type is *readable* if all its instances can be created through the Common Lisp reader using the

[2] ***standard-input***, ***standard-output***, ***error-output***, ***trace-output***, ***query-io*** and ***debug-io*** are all initially bound to synonym streams that forward all operations to the stream bound to ***terminal-io***.

Transmissible	number, character, symbol, list, simple-vector[a], simple-array, simple-bit-vector, simple-string, uncompiled function, hash-table, random-state, pathname, structure
Non-Transmissible	compiled function, stream, package, readtable, CLOS instances[b]

Table 1: Transmissible and Non-Transmissible Types

[a]Simple arrays, vectors, etc. have no fill pointers, can not dynamically adjust their size, and are not "displaced to" another array.

[b]Programmers may define `marshall` and `unmarshall` methods for any class they wish to transmit.

type's default print representation. Some examples include `simple-arrays` and `lists`. `Functions` and `hash tables` are examples of non-readable, but transmissible, types.

In most cases, readable types are trivially transmissible. From one evaluator we simply pass an object's print representation and at the other evaluator we reconstitute a copy of the object using the built-in `read` function. While both symbols and structures are readable, neither type is trivially transmissible.

5.1.1. Symbols

A symbol is composed of a symbol name, package name and a property list. When the Common Lisp reader parses a form, any token thought to be a symbol name is looked up in the current package (assuming no package qualifier is given). If the name is found, the associated symbol is returned. Otherwise, a new symbol is created.

In Avalon/Common Lisp, a symbol's package name is part of its transmissible representation. Transmitted symbols are interned in an identically-named package on the remote evaluator. Uninterned symbols, which have no package association, are never interned.

All of this combines to produce some unexpected results. Let's assume the database, set, and printing and matching functions for our bibliography server have been loaded into one evaluator and the appropriate functions exported from the `bib` package. A user at another evaluator might query the server as follows:

```
(remote
  (bib:print-bib-entrys
    (bib:matches bib:author-named-Butler)))
```

Evaluation of this form typically produces one of two errors; either the `bib` package does not

exist or at least one of the symbols cannot be found, since it has not been exported from the **bib** package. These errors are signaled by the initiating evaluator as it marshalls the request. While the **bib:print-bib-entrys** expression is not evaluated by the initiating evaluator, the expression itself must be built. The errors result from the read-time symbol lookups. As a second attempt, the user might try:

```
(remote
  (let ((*package* 'bib))
    (print-bib-entrys
      (matches author-named-Butler))))
```

This too will fail, but this time on the remote evaluator. When the initiating evaluator packages up the **print-bib-entrys** symbol for transmission, it includes the package name, which may or may not be **bib**. Let's assume it is **user**. When the form is evaluated by the remote evaluator, it looks for **print-bib-entrys** in the **user** package, not **bib**, resulting in an undefined function error.

To avoid these problems, we recommend that the export list for a server be placed in a separate file and loaded into any evaluator that may act as a client or server. For our bibliography example, this file includes:

```
(in-package 'bib)
(export
 '(union intersection difference matches print-bib-entrys))
```

The client may now use either of the following to execute his query successfully.

```
(remote
  (bib:print-bib-entrys
    (bib:matches bib:author-named-Butler)))

(use-package 'bib)
(remote
  (print-bib-entrys
    (matches author-named-Butler)))
```

Note that no evaluator can make assumptions about the value of another evaluator's current package.

A symbol's property list is considered to be part of a process' global environment and is not transmitted.

5.1.2. Structures

Structures pose a problem in that accompanying any structure definition is a collection of functions that operate on it. The **remote** facility only transmits a representation of an object. It does not

guarantee that a structure's accessor or constructor functions are defined on the remote evaluator. The programmer must ensure that structure types are defined on each evaluator that may operate on an instance of the structure. As with symbols, we recommend that any relevant **defstructs** be placed in a file that is loaded into both client and server evaluators.

5.2. Example Uses of Extensions

A programmer may wish to define his own transmission functions for several reasons - to provide a more efficient representation, to take advantage of application-specific knowledge, or to make an unsupported type transmissible. As before, our bibliography application provides an example.

The database represents annotations by structures having three slots. **Bibid** contains an identifier that pairs an annotation with its corresponding bibliography entry; **access** may be either public or private and controls who sees the annotation; **note** contains the actual text. Avalon/Common Lisp's default transmission support has no knowledge of annotations and would ship the complete object, even if access is private. Hence, we need to supply our own marshall routines.

```
(defstruct annotation
  bibid
  access
  note)

(defun marshall-annotation (annote stream)
  (let ((access (annotation-access annote)))
    (cond ((eql access 'private)
           (marshall-object nil stream))
          (t (marshall-object (annotation-bibid annote) stream)
             (marshall-object (annotation-access annote) stream)
             (marshall-object (annotation-note annote) stream)))))

(defun unmarshall-annotation (stream)
  (let ((bibid (unmarshall-object stream)))
    (if bibid
        (make-annotation :bibid bibid
                         :access (unmarshall-object stream)
                         :note (unmarshall-object stream))
        nil)))

(defmarshall 'annotation 'marshall-annotation 'unmarshall-annotation)
```

Marshall-annotation simply checks the access field and invokes Avalon/Common Lisp 's standard marshalling function to package up the appropriate value(s). Unmarshall-annotation reverses the process. The **defmarshall** call notifies Avalon/Common Lisp that **marshall-annotation** and **unmarshall-annotation** should be used to encode and decode annotations.

Transmit supplies **marshall** and **unmarshall** methods for defining transmission functions for CLOS instances. While the **defmarshall** form could be used, the **marshall** methods provide a mechanism more consistent with an object-oriented style. Methods cannot be used exclusively because of the poor integration of CLOS classes with the standard Common Lisp type system.

When a type is defined with **deftype**, no associated class is created.

Since CLOS instances are not transmissible by default, users must define methods for each class they wish to transmit. In the simplest case, **marshall** uses previously defined transmission functions to package up each of the instance's slots. For example,

```
(defclass bib-entry ()
  ((author :initform nil :initarg :author)
   (title :initform nil :initarg :title)
   (publisher :initform nil :initarg :publisher)
   (year :initform nil :initarg :year)))

(defmethod marshall ((entry bib-entry) stream)
  (with-slots (author title publisher year) entry
    (marshall-object 'bib-entry stream)
    (marshall-object author stream)
    (marshall-object title stream)
    (marshall-object publisher stream)
    (marshall-object year stream)))

(defmethod unmarshall ((code (eql 'bib-entry)) stream)
  (let* ((author (unmarshall-object stream))
         (title (unmarshall-object stream))
         (publisher (unmarshall-object stream))
         (year (unmarshall-object stream)))
    (make-instance 'bib-entry :author author :title title
                              :publisher publisher :year year)))
```

Note that the first item **marshall** packages up, the symbol **bib-entry**, must match the parameter specializer of unmarshall's first argument, **(eql 'bib-entry)**. Like structures, classes have associated methods that operate on them. The programmer must ensure that all relevant **defclasses** and **defmethods** are loaded on each evaluator that operates on instances of the class.

5.3. Exported Forms

The **transmit** package provides two mechanisms for defining new transmission functions, **defmarshall** and paired **marshall** and **unmarshall** methods.

transmit:defmarshall *type marshall-fn unmarshall-fn* [*Macro*]

This macro defines how objects satisfying the type specifier, *type*, are to be transmitted. *marshall-fn* names a function which takes two arguments, an object and a stream, and appends an encoded representation of the object to the stream. *Unmarshall-fn* reverses the process. It takes a single argument, stream, from which it reads the encoded representation. This representation is decoded and an object is built and returned.

Defmarshall adds the new definition to a partially ordered list of user-defined transmission functions and returns this updated list. A continuable error is signaled if *type* already has a marshall definition.

When an object is transmitted, the most specific (as determined by **subtypep**), available marshall function is called to do the encoding. **Subtypep** cannot always determine the relationship between two types, particularly when a type specifier contains a satisfies clause. Defmarshall issues the warning:

Warning: Uncertain about precedence ordering between *type1* **and** *type2*.

if it cannot definitively order the marshall functions. The programmer should check that *type1* and *type2* are disjoint. There is currently no mechanism for supplying hints to defmarshall as to how to resolve ambiguous type relationships. In the worst case, the programmer must sequence his **defmarshall** calls carefully, but this scenario should not arise.

When defining a new transmission function, it may be helpful to use existing representations for some components of the type. The following functions are used for this purpose.

transmit:marshall-object *object stream* [*Function*]

This function appends a transmissible representation of *object* to *stream*. The partially ordered list of user-defined marshall functions is searched first. If no match is found, (by applying the **typep** predicate to *object* and the user-provided type specifiers) one of Avalon/Common Lisp's default functions is called. Sharing is preserved within the transmitted object. An error is signaled if no transmissible representation exists.

transmit:unmarshall-object *stream* [*Function*]

Stream contains a representation constructed by a call to **marshall-object**. This function creates and returns a new object built from this representation.

transmit:lookup-marshall-def *type* [*Function*]

This function returns a list containing the names of the user-defined marshall and unmarshall functions for *type*. **Nil** is returned if no definition is found.

transmit:delete-marshall-def *type* [*Function*]

This function removes the marshall definition for *type* and returns the new list of user-defined transmission functions. An error is signaled if no definition is found for *type*.

An alternate mechanism is provided for defining transmission functions for instances of user defined classes. The programmer provides a pair of methods for the class that satisfy the following definitions.

transmit:marshall *instance stream* [*Method*]

This method appends two items to *stream*, a code and a transmissible version of *instance*. The code controls which unmarshall method is dispatched to on the remote evaluator.

transmit:unmarshall *code stream* [*Method*]

This method builds and returns a new CLOS instance from the representation in *stream*.

6. Persistence

Avalon/Common Lisp supports persistence through declarations of persistent bindings. Since it makes sense to access recoverable objects only when executing a transaction, we also provide control primitives to begin, commit and abort transactions (Section 7). We begin by showing how these features might be used by our bibliography database server.

6.1. Example Uses of Extensions

We start by making the database's binding persistent and initializing it:

```
(defpersistent $bib-database$ (make-persistent (empty-set)))
```

Make-persistent creates a recoverable object; **defpersistent** defines **$bib-database$** as a binding to be recoverable, and creates a binding between **$bib-database$** and the recoverable empty set. (By convention, we use the "$" characters to distinguish those symbols used for persistent bindings from those used for volatile ones.)

We use transactions for standard database operations such as adding, modifying, and deleting entries. Consider the function for adding a bibliography entry:

```
(defun add-bib-entry (entry)
  (with-transaction
   (if (valid-bib-entry-p entry)
       (adjoin $bib-database$ (make-persistent entry))
       (abort-transaction 'invalid-bib-entry))))
```

If the entry is valid, i.e., well-formed and not already in the database, then we make the volatile value of the **entry** argument persistent and add it to the database. Otherwise, we abort the transaction signaling the abort condition **invalid-bib-entry**. Since the update is done within a transaction, if a crash occurs during the update, the state of the bibliography database will be as if the update never occurred; Camelot's recovery algorithm will guarantee the database is restored to a previously saved consistent state.

The counterpart to **make-persistent** is **make-volatile**. Since an evaluator communicates with a recoverable store, retrieving a persistent binding from it gives us a handle on a recoverable object. Upon retrieval, we are free to continue to use the object as a recoverable object until we need to either call a standard Common Lisp function or transmit the object back to the local evaluator. Thus as a server writer, we have some latitude as to when we make the **make-volatile** call. For

rec-eq	rec-null	rec-list	rec-elt
rec-eql	rec-car	rec-nth	rec-length
rec-equal	rec-cdr	rec-svref	rec-symbol-name
rec-equalp	rec-cons	rec-vector	rec-symbol-package-name

Table 2: "Recoverable" Versions of Standard Lisp Functions

fixnum	symbol	simple-string
single-float	list	simple-vector
double-float	structure	

Table 3: Recoverable Types

example, both **print-bib-entrys** below have the same eventual effect:

```
(defun print-bib-entrys ()
  (set-mapc #'(lambda (set-entry)
                (print-bib-entry (make-volatile set-entry))) $bib-database$))

(defun print-bib-entrys ()
  (set-mapc #'print-bib-entry (make-volatile $bib-database$)))
```

In the first version, **set-mapc** operates on a persistent set (and uses **rec-car**, **rec-cdr**, etc. to traverse the **$bib-database$**). In the second, **set-mapc** operates on a volatile object. **Make-persistent** and **make-volatile** are each idempotent and are inverses of each other.

The **rstore** package exports "recoverable" versions of some standard Lisp functions (see Table 2). These functions operate on objects retrieved from recoverable store, rather than normal non-recoverable Lisp objects.

Table 3 lists those Common Lisp built-in types which currently have recoverable versions. Structures must be defined as recoverable if persistent instances are to be created. This is accomplished by using an extended version of the **defstruct** macro.

In the example below, we include the **recoverable** option in our structure definition, indicating that both volatile and recoverable instances may be created. **Defstruct** generates two constructor functions, **make-book-entry** and **make-recoverable-book-entry** that create volatile and recoverable instances respectively, and supplies accessor functions and setf methods that work on both instance types.

The second expression builds a recoverable instance, while the third constructs a recoverable instance from a volatile one. The last expression simply updates the value of entry's publisher slot.

```
(defstruct (book-entry :recoverable)
  (author nil)
  (title nil)
  (publisher nil)
  (year nil))

(make-recoverable-book-entry
  :author "H. Abelson and G.J. Sussman"
  :title  "Structure and Interpretation of Computer Programs"
  :publisher "MIT Press"
  :year 1985)

(make-persistent
  (make-book-entry
    :author "H. Abelson and G.J. Sussman"
    :title  "Structure and Interpretation of Computer Programs"
    :publisher "MIT Press"
    :year 1985))

(setf (book-entry-publisher entry)
      (make-persistent "Springer Verlag"))
```

6.2. Exported Forms

The **rstore** package exports forms to define persistent bindings and construct and access persistent objects while the **trans** package provides transaction support (Section 7).

rstore:defpersistent *variable* [*initial-value* [*documentation*]] [*Macro*]

This form is similar to the **defvar** form, except that any binding to *variable* is recoverable, i.e., survives crashes and supports failure atomicity. If given, *initial-value* is assigned to *variable*, as long as *variable* has not previously been bound. *Initial-value* must evaluate to a recoverable object and is only evaluated if it is used to initialize the binding.

All subsequent **setq** operations to *variable* will change the binding atomically; **setq** operations to persistent variables can be aborted if evaluated within a transaction.

rstore:make-persistent *object* [*Function*]
rstore:make-volatile *object* [*Function*]

These functions create a persistent (volatile) representation of *object*. If *object* is already persistent (volatile), it is returned as the result.

rstore:defstruct *name-and-options* [*doc-string*] {*slot-description*} [*Macro*]

This macro extends the standard **defstruct** macro to include definitions of recoverable structures. A new option, *recoverable*, indicates that recoverable instances may be created. The standard constructor function, **make-***name*, builds volatile instances; a new constructor function, **make-recoverable-***name* builds recoverable ones. Accessor functions and setf methods may be applied to both volatile and recoverable instances.

7. Failure Atomicity

The **trans** package supplies forms for beginning, committing and aborting transactions. Placing the transaction forms in a separate package permits evaluators without recoverable stores to group sets of server operations into atomic actions. Sample uses are given in Section 6.1.

7.1. Exported Forms

The exported forms include:

trans:with-transaction *body* [*Macro*]
trans:with-top-level-transaction *body* [*Macro*]

Both forms initiate a new transaction and evaluate *body*. **With-transaction**, if evaluated dynamically within another transaction, will begin a nested transaction; otherwise it starts a top-level transaction. **With-top-level-transaction** always initiates a new top-level transaction. Both forms return a multiple value consisting of a status signifying whether or not the transaction committed, and the result of the last expression in *body*.

Normal evaluation of either form results in a committed transaction. Exceptional exits from the *body* (via **catch/throw** and local exits) result in the transaction aborting. Transactions can also be explicitly aborted via use of **abort-transaction**.

trans:abort-transaction *retval* **&optional top-level** [*Macro*]

This form aborts the currently executing transaction. If the optional argument is nil (the default), the innermost dynamically nested transaction is aborted and the value of *retval* is returned as the status in the multiple-value result of **with-transaction**. Otherwise, the current (dynamically scoped) top-level transaction is aborted.

8. Implementation Status

Several features described in Clamen et al., [1] are not implemented. No callback support (via the **local** and ***local-evaluator*** forms) is provided. Side effects on objects contained in the transmitted (lexical) environment are not propagated across evaluator boundaries. Only minimal

facilities for registering and locating evaluators are available. No mechanism exists for locating an evaluator by specifying a set of attributes it must possess and no **make-evaluator** function is provided. Existing evaluators can be found, but remote evaluators cannot be created automatically.

Currently, a complete copy of an object, that is, its transitive closure, is transmitted. Partial transmission of large objects or transmission of partially evaluated objects as described in [1] is not supported. One version of the bibliography application uses its own marshalling functions to provide partial transmission of highly networked, readable objects. It transmits an object's root and its descendants up to n-levels deep.

A Running Avalon/Common Lisp at CMU

A1. Environment

Avalon/Common Lisp runs on IBM RT's and is implemented on top of CMU Common Lisp and Camelot. A Lisp process' recoverable storage is managed by a Camelot data server, hereafter called the Lisp recoverable object or *lrec* server [4].

A2. Paths

The following lines should be added to your .login file.

```
setpath -ib /usr/cs /afs/cs/project/avalon-1/cam_beta
setpath PATH -i /afs/cs/project/avalon-1/acl/ky/cam/nca_tools
setenv LREC_SERVER "/afs/cs/project/avalon-1/acl/stable/cam/bin/lrec"
```

The first command adds the Camelot directories to your path, the second adds a directory containing tools for automatically starting the lrec server and shutting down Camelot. The **$LREC_SERVER** environment variable specifies the complete pathname of the lrec server to be used.

A3. Starting the Lrec Server

Camelot must be started first. If you want to start with a clean world, include the **-init** switch in the command line. Camelot will then ask whether the logger and paging areas should be initialized. Answering yes to both questions clears the old data from the system. Otherwise, Camelot will restart all servers that were running the last time it went down. In the sequences that follow, user input is shown in italics.

```
% camelot -init
log: Initialize the Logger? (yes or no): [no] y
disk: Initialize the Paging area? (yes or no): [yes]


   Sending output to /usr/camelot/log/camelot.out.

%
Camelot release 0.99(75) [beta] initializing.
Camelot compiled Thu Nov 16 13:50:47 1989 on KY.AVALON.CS.CMU.EDU.
```

Once Camelot is running, the lrec server can be started.

```
% start_lrec
server: 2 owner: admin auto-restart: no
segment: 2 quota: 10 chunks
cmdline: /afs/cs/project/avalon-1/acl/stable/cam/bin/lrec
```

To shut down both Camelot and the lrec server, simply enter

% *cam_shutdown*

A4. Loading Avalon/Common Lisp

Avalon/Common Lisp uses the Ergo Box facility to manage its load and compile time file dependencies. From a running Lisp, enter the following:

```
(load "/afs/cs/project/avalon-1/acl/stable/adm/acl-init")
```

This loads the necessary pieces of the box facility as well as the box definitions for the Avalon/Common Lisp components. You are now ready to load and use whichever pieces of Avalon/Common Lisp you need.

In each of the following, include the **boxload** forms in the file that loads the application and the **use-package** forms in the application code.

To use remote evaluation:

```
(box:boxload "remote-eval")
(use-package 'remote-eval)
```

To define your own transmission functions:

```
(box:boxload "transmit")
(use-package 'transmit)
```

To use transactions:

Start Camelot and then enter these forms.

```
(box:boxload "trans")
(use-package 'trans)
```

To use both remote evaluation and transactions:

This combination is most often used by client evaluators. The **remote-eval+trans** box simply loads the **remote-eval** and **trans** boxes plus a bit of glue. Start Camelot before executing the following:

```
(box:boxload "remote-eval+trans")
(use-package 'remote-eval)
(use-package 'trans)
```

To use the recoverable store:

Start Camelot and the lrec server and then execute the following:

```
(box:boxload "rstore")
(shadow '(setq defstruct))
(shadowing-import '(rstore:setq rstore:defstruct))
(use-package 'rstore)
```

To use everything:

Start Camelot and the lrec server and then enter:

```
(box:boxload "avalon")
(shadow '(setq defstruct))
(shadowing-import '(avalon:setq avalon:defstruct))
(use-package 'avalon)
```

This is equivalent to loading both the **remote-eval+trans** and **rstore** boxes.

A5. Example Programs

Directory **/afs/cs/project/avalon-1/acl/stable/examples** contains several small example programs. Each example is located in a separate subdirectory that includes a **README** file containing instructions for executing the program.

References

[1] S.M. Clamen, L.D. Leibengood, S.M. Nettles, J.M. Wing, *Reliable Distributed Computing with Avalon/Common Lisp*, Technical Report CMU-CS-89-186, September 1989.

[2] Stewart M. Clamen, *Towards Avalon/Lisp – Remote Lisp Evaluation*, Avalon Note 14, June 1989.

[3] Eric C. Cooper and Richard P. Draves, *C Threads*, Technical Report CMU-CS-88-154, June 1988.

[4] Jeffrey L. Eppinger, Lily B. Mummert, Alfred Z. Spector, editors, *Guide to the Camelot Distributed Transaction Facility including the Avalon Language*, Draft of September 18, 1989.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, *The Notion of Consistency and Predicate Locks in a Database System*, Communications ACM, Vol. 19, No. 11, November 1976, pp. 624-633.

[6] Guy L. Steele Jr., *Common LISP: The Language*, Digital Press, 1984.